Course notes for High Performance Programming and Systems

Troels Henriksen, David Gray Marchant, Kenneth Skovhede

January 23, 2024

0.1 Meta

These notes are written to supplement the textbooks in the course *High Performance Programming and Systems*. Consider them terminally in-progress. These notes are not a textbook, do not cover the entire curriculum, and might not be comprehensible if isolated from the course and its other teaching activities.

These notes are made available under the terms of the *Creative Commons Attribution-ShareAlike 4.0 International Public License*.

Contents

0.1 Meta . . . 1

Contents **2**

Definitions **4**

1 Computer Systems **6**

2 Data as Bits **8**
2.1 A Bit . . . 8
2.2 Boolean Logic . . . 11
2.3 Bit Arithmetic . . . 13
2.4 Signed Numbers . . . 17

3 Floating-Point Numbers **22**
3.1 The Basic Problem . . . 22
3.2 Fixed-point Fractional Numbers . . . 23
3.3 Intuition behind floating-point numbers . . . 25
3.4 IEEE 754 . . . 26
3.5 Rounding . . . 30
3.6 Arithmetic . . . 32

4 Data as Bytes **34**
4.1 Memory . . . 35
4.2 Multi-byte Words . . . 35
4.3 Text and Characters . . . 37

5 Compiled and Interpreted Languages **40**
5.1 Low-level and High-Level Languages . . . 40
5.2 Compilers and Interpreters . . . 41
5.3 Tombstone Diagrams . . . 46
5.4 Combining Python and C . . . 51

6 Data Layout **53**
6.1 Arrays in C . . . 53

7 Locality and Caches **59**

7.1 Locality of Reference 59
7.2 Memory Hierarchies 61
7.3 Cache Organisation 64
7.4 Cache Performance 67

8 Operating Systems **72**
8.1 The Kernel 72
8.2 Processes 73
8.3 Virtual Memory 74

9 Networks **77**
9.1 OSI layers 77
9.2 Physical layer 78
9.3 Link layer 79
9.4 Network layer 80
9.5 Transport layer 82
9.6 Application layers 87
9.7 Network Programming 89

10 OpenMP **91**
10.1 Basic use of OpenMP 91
10.2 Reductions 95
10.3 Nested loops 96
10.4 Scheduling 98
10.5 The Big Concurrency Problems 100

11 Parallel Speedup and Scalability **106**
11.1 Speedup 106
11.2 Scalability 108

12 Loop Dependence Analysis **112**
12.1 Direction vectors 113
12.2 Determining loop parallelism 121

13 Loop Transformations **123**
13.1 Loop interchange: legality and applications 123
13.2 Loop distribution: legality and applications 125
13.3 Eliminating false dependencies (WAR and WAW) 128
13.4 Loop stripmining, block and register tiling 131

Bibliography **135**

Definitions

2.1 Definition (Bit) 8
2.2 Definition (Bit vector) 9
2.3 Definition (Concatenation of bit vectors) 9
2.4 Definition (Bit vector to natural number) 10
2.5 Definition (Natural number to w-bit vector) 10
2.6 Definition (Word) 12
2.7 Definition (Bitwise operations on words) 13
2.8 Definition (Integer addition) 14
2.9 Definition (Integer overflow) 14
2.10 Definition (Logical left-shift by k bits) 15
2.11 Definition (Logical right-shift by k bits) 16
2.12 Definition (Integer multiplication) 16
2.13 Definition (Bit vector to integer using sign-magnitude) 17
2.14 Definition (Integer to Two's Complement) 18
2.15 Definition (Two's Complement to Integer) 18
2.16 Definition (Negating a Two's Complement number) 19
2.17 Definition (Integer subtraction in Two's Complement) 20
2.18 Definition (Arithmetic right-shift by k bits) 20

3.1 Definition (Fixed-point number) 24
3.2 Definition (Bit vector interpreted as unsigned fixed point) 24
3.3 Definition (Representation of floating-point number x) 28
3.4 Definition (Fields of a floating-point number x) 28
3.5 Definition (Correct rounding) 31
3.6 Definition (Floating-point overflow) 32
3.7 Definition (Floating-point underflow) 32

4.1 Definition (Big Endian) 36
4.2 Definition (Little Endian) 36
4.3 Definition (Coded character set) 37

7.1 Definition (Principle of locality) 59
7.2 Definition (Temporal locality) 60
7.3 Definition (Spatial locality) 60
7.4 Definition (Memory hierarchy) 62
7.5 Definition (Cache) 62

7.6 Definition (Cache hit) 63
7.7 Definition (Cache miss) 63
7.8 Definition (Cache block) 63
7.9 Definition (Working set) 64
7.10 Definition (Memory footprint) 64
7.11 Definition (Compulsory miss) 64
7.12 Definition (Capacity miss) 64
7.13 Definition (Cache line) 65
7.14 Definition (S-way set associative cache) 65
7.15 Definition (Conflict miss) 66
7.16 Definition (Direct-mapped cache) 67
7.17 Definition (Fully associative cache) 67
7.18 Definition (Miss rate) 67
7.19 Definition (Hit time) 68
7.20 Definition (Miss penalty) 68

10.1 Definition (Concurrency) 95
10.2 Definition (Parallelism) 95
10.3 Definition (Static scheduling) 98
10.4 Definition (Dynamic scheduling) 98
10.5 Definition (Race Condition) 100
10.6 Definition (Deadlock) 102

11.1 Definition (Speedup in latency) 107
11.2 Definition (Speedup in throughput) 108
11.3 Definition (Strong scaling) 108
11.4 Definition (Weak scaling) 108
11.5 Definition (Amdahl's Law) 109
11.6 Definition (Gustafson's Law) 110

12.1 Definition (Loop Dependency) 115
12.2 Definition (Dependency direction vector) 118
12.3 Definition (Dependency direction matrix) 121

13.1 Definition (Dependency graph) 125

Chapter 1

Computer Systems

There is nothing a computer can that a human cannot also do. The only advantage that computers offer is far greater speed. As the machines of the industrial revolution amplified the physical power of humanity beyond what our bodies are capable of, so do the machines of the computer revolution amplify our computational power beyond what is possible using only our brains. Therefore, while it is a common trope that "programmer time is more valuable than computer time", and that even inexpertly written programs are *fast enough*, ultimately it is execution speed that is the reason why computers are important and interesting.

In the chapters that follow, we will look at how modern computers work and how to construct programs that run fast. We will take a fairly high level view—there are many details that will only be covered briefly, and reading this text will not make you an expert programmer. However, it will hopefully give you an appreciation of how to design programs that are not *accidentally* inefficient, and where to start looking when you have a program that works correctly, but is too slow to be useful. This focus on *performance* is the underlying theme of the text. You've been hopefully been taught how to write *correct* programs; now is the time to make them fly.

Our initial focus will be on understanding the distinction between *representation* and *interpretation*. Computers know nothing of images, graphs, sounds, text, files, networks, humans, or even—when you really get down to it—numbers. Whenever we want to process and transform data that is meaningful to humans, we first have to represent it in a way that the machine can understand, which usually boils down to sequences of bits. How we design such encodings has significant performance and convenience implications. While this view will be everpresent throughout the course, it is especially the focus of chapters 2 to 4.

Not only do computers not comprehend most data that is meaningful to humans, they also do not understand human commands. Instead they follow orders encoded as *machine code*, which can be seen as a particularly human-hostile programming languages. In chapter 5 look at how computers are made

accessible to humans through programming languages, look at the various ways we can categorise and classify them, and what the implications might be for performance.

For modern computers, sheer computational power—e.g. how fast we can multiply two numbers—is rarely the main performance bottleneck. Indeed, it is often far more costly to retrieve data than it is to operate on it once it has been fetched. In chapters 6 and 7 we will look at how larger collections of data can be structured and what the implications are for performance. In particular, the notion of *locality* is one of the most important concerns for program efficiency.

Beyond locality, another crucial technique necessary to obtain good performance is *parallelism*. Modern computers are able to perform multiple operations simultaneously, and if we write programs that do not take advantage of this capability, we are in effect not exploiting the computer to its full extent. Chapters 10 and 11 discuss programming and measurement techniques for parallel programming.

Chapter 2

Data as Bits

A computer is a machine for processing and transforming information. Machines inevitably must operate on things that physically exist, so in order to process information in a machine, we must *represent* the information in some physical way. While we stop short of discussing precisely the physical phenomena that underlie modern computers, we will look at the notion of *value encodings*—how mathematical objects can be represented such that they can be processed by machines.

2.1 A Bit

Definition 2.1 (Bit) *A bit (*binary digit*) is a logical state that can represent two possible values, which we write as* 1 *or* 0*.*

Conventionally, and in this text, the two possible values are written 1 and 0, as a reference to their interpretation as numbers, but this merely a question of notation. We could equally well have used *true*/*false*, *a*/*b*, *yes*/*no*, or any other notation that allows us to distinguish unambiguously between the two possible values. By convention, we say that a bit is *set* when 1 and *unset* when 0.

Bits are used in information theory as a unit of information. For computers, bits are convenient because any physical phenomenon that can be interpreted as having two states can be used to represent a bit. For example:

1. High or low voltage in an electrical wire.
2. Absence or presence of a hole in some material.
3. Vertical or horizontal polarity of light.
4. Heads or tails of a coin.
5. Whether a cup is full or empty.
6. Whether a corridor is full of soldier crabs or not [2].

Some of these representations are more practical than others, but all are ultimately based on the notion of a *bit*. The choice of which representation is most practical in a given setting is largely based on how we can construct machinery that manipulates the physical representation of the bits. Modern computers are overwhelmingly electronic, and use *transistors* to transform electrical signals representing bits, but optical representations are also common for long-distance communications.

2.1.1 Bit Vectors

Single bits rarely occur in isolation. Typically we use a sequence of bits, called a *bit vector*.

Definition 2.2 (Bit vector) *A bit vector x of length w is an ordered sequence of w bits, which we write as $x = \langle x_{w-1} \ldots x_0 \rangle$.*

Note the convention that bit x_0 in a bit vector is written at the *rightmost* position. This resembles the ordering of digits in mathematical notation.

Definition 2.3 (Concatenation of bit vectors) *Two bit vectors juxtaposed is interpreted as concatenation and defined as follows:*

$$\langle x_{n-1} \ldots x_0 \rangle \langle y_{m-1} \ldots y_0 \rangle = \langle x_{n-1} \ldots x_0 \; y_{m-1} \ldots y_0 \rangle$$

Bit vectors can be interpreted as encoding various mathematical objects. We will initially be representing values that look very similar to bits, and the following may seem unnecessarily long-winded and ceremonious, but eventually we will look at more complicated encodings. The goal is to establish a firm distinction between the *encoding* of some mathematical object (say, a number) as a bit vector, and the mathematical object itself. Such an encoding is defined by specifying *conversion functions* between the set of w-bit vectors, which we denote $\mathbb{B}^w$, and the mathematical set we wish to encode, say the natural numbers $\mathbb{N}$.

2.1.2 Bit Vectors as Natural Numbers

One of the most obvious ways to interpret a bit vector is as a number in base 2. For example, the bit vector $\langle 1001 \rangle$ might be interpreted as an encoding of the number $1001_2 = 9_{10}$. Note the subscripts used to denote the *radix* of the literals—we will include these whenever the radix would otherwise be ambiguous. However, it is important to note that a $\langle 1001 \rangle$ *is not the same* as 1001_2! They are objects in completely different domains, and it makes no sense to say that they are equal or unequal. It is merely a quirk of notation that they look similar when written down, and we shall soon enough see encodings where this is not the case.

To completely specify the encoding of natural numbers, we must define how a bit vector of length w is interpreted as a number. Here we do treat single bits as numbers, 0 or 1, and multiply them with a *weight*.

Definition 2.4 (Bit vector to natural number)

$$\text{Bits2N}(\langle x_{w-1} \cdots x_0 \rangle) := \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Example 2.1 (Interpreting $\langle 1101 \rangle$ as natural number)

$$\begin{aligned} \text{Bits2N}(\langle 1101 \rangle) &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \\ &= 13 \end{aligned}$$

The conversion of a number to a bit vector is slightly less pleasant, and is defined by the following recursive procedure.

Definition 2.5 (Natural number to w-bit vector)

$$\text{N2Bits}_1(0) := \langle 0 \rangle \tag{2.1}$$

$$\text{N2Bits}_1(1) := \langle 1 \rangle \tag{2.2}$$

$$\text{N2Bits}_w(x) := \begin{cases} \langle \text{N2Bits}_{w-1}(\lfloor \frac{x}{2} \rfloor) \rangle \langle 0 \rangle & x \textit{ is even} \\ \langle \text{N2Bits}_{w-1}(\lfloor \frac{x}{2} \rfloor) \rangle \langle 1 \rangle & x \textit{ is odd} \end{cases} \tag{2.3}$$

Note that $\text{N2Bits}_w(x)$ always produces a bit vector with w bits, no matter the magnitude of x. This is because we tend to work only with bit vectors of some fixed size. We will return to this in section 2.2.2.

Example 2.2 (Encoding 9 as bit vector)

$$\begin{aligned} \text{N2Bits}_4(9) &= \text{N2Bits}_4(2 \cdot 4 + 1) && (2.4) \\ &= \text{N2Bits}_3(4)\langle 1 \rangle && (2.5) \\ &= \text{N2Bits}_2(2)\langle 1 \rangle\langle 1 \rangle && (2.6) \\ &= \text{N2Bits}_1(1)\langle 0 \rangle\langle 0 \rangle\langle 1 \rangle && (2.7) \\ &= \langle 1 \rangle\langle 0 \rangle\langle 0 \rangle\langle 1 \rangle && (2.8) \\ &= \langle 1001 \rangle && (2.9) \end{aligned}$$

This representation can express only non-negative numbers, and is therefore called *unsigned*, because there can be no leading minus sign. In section 2.4 we will discuss negative numbers.

When $x \geq 2^w$, then x cannot be encoded with a w-bit vector by definition 2.5. Intuitively, x is *truncated* and only the lower w bits of its "full" representation is included. This is an example of *overflow*, which we will return to in section 2.3.1.1. If we wished, we could also have designated a single distinct bit pattern to encode all numbers that otherwise have no encoding, which would let us detect anomalous cases. We will see an example of such an encoding in chapter 3.

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$	$\neg p$
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

Figure 2.1: Truth table for *and*, *or*, *exclusive-or*, and negation.

Having an encoding of numbers is not terribly useful unless we can also perform *operations*, such as arithmetic, on numbers represented in the given encoding. However, before we can do that, we have to talk about Boolean logic.

2.2 Boolean Logic

While numbers are one of the most interesting things we can encode as bit vectors, we will start out by looking at bits as truth values. *Truth* as a subject of computation was investigated by the English mathematician George Boole (1815-1864), whose Boolean logic far predates the modern notions of computers and bits.

Just as we can define operators on *numbers* (such as addition), we can define operations on *truth values*, writing T for truth and F for falsity. For example, *logical-and*, written $p \wedge q$, is true if p and q are both true, and otherwise false, while *logical-and*, written $p \vee q$, is true if either operand is true. The *exclusive-or* operation $p \oplus q$ is true if exactly one of p and q is true, and *logical negation* $\neg p$ is true only if p is false. This is shown in fig. 2.1.

Since a binary boolean operator can have two possible results (T or F) for a given combination of operands, and there are four possible combinations of operands, there are $2^4 = 16$ distinct binary operators on booleans. Most of these can be written in terms of other operators. For example, the *negated-and* operator can be defined as

$$\text{nand}(p, q) := \neg(p \wedge q) \tag{2.10}$$

Interestingly, it turns out that the nand operation is universal, in that *any* boolean function can be written using a combination of nand operations. Examples:

$$\neg p = \text{nand}(p, p) \tag{2.11}$$
$$p \wedge q = \text{nand}(\text{nand}(p, q), \text{nand}(p, q)) \tag{2.12}$$
$$p \vee q = \text{nand}(\text{nand}(p, p), \text{nand}(q, q)) \tag{2.13}$$

2.2.1 Boolean Operations as Bits

The truth values of Boolean logic can easily be encoded as bits - by treating 1 as T and 0 as F, we can apply the boolean operators directly:

$$0 \wedge 1 = 0 \tag{2.14}$$
$$0 \vee 1 = 1 \tag{2.15}$$
$$0 \oplus 1 = 1 \tag{2.16}$$
$$\neg 0 = 1 \tag{2.17}$$

In section 2.1 we remarked that bits can be represented using any physical phenomenon capable of two distinct states. In order to be practical for *computation*, we must also be able to easily implement boolean operations on pairs of bits. As mentioned above, the nand operation is universal, so if we can show how to implement it, we can implement any boolean function.

In practice, it turns out that representation of bits as high and low voltages makes it easy to implement logical operations with *transistors*. It is relatively straightforward to create a *gate* that has two input wires and one output wire, where the voltage of the output wire depends on the voltages of the input—and these gates can be made extremely small using modern manufacturing techniques. This is why electronic computers have become the most popular way of implementing computation.

We will not delve further into how logical operations are physically implemented. Instead, we will use the logical operators to build ever more elaborate operations on bit vectors, knowing that as long as we can express a computation in terms of bit operations, we can ultimately express it in hardware. On this humble foundation we will build ever more elaborate data representations.

2.2.2 Words

One important concession to practicality is that we will operate on bit vectors of fixed lengths. While it is possible to build computers that operate on bit vectors of arbitrary lengths, it is much more efficient to build circuits that operate on a fixed number of bits at a time. Usually these are powers of two—$8, 16, 32, 64$, etc. In the computer systems nomenclature, a bit vector of some directly hardware-supported fixed size is called a *word*. Generally, when we use the term *w-bit word*, we mean a bit vector containing w bits.

Definition 2.6 (Word) *A word is a bit vector of some fixed size w, on which the computer can efficiently operate directly.*

A w-bit word can express 2^w different permutations of bits, meaning it can represent 2^w different values. When deciding on a value representation, deciding how to best exploit this limited range is important. In definition 2.4 we decided that a w-bit word can represent integers in the range $[0, 2^w - 1]$, which certainly feels intuitive, but we could just as well have defined a conversion

function that encoded integers in the range $[b, b+2^w-1]$ for some b[1]. We will also have to decide what happens when an operation produces a value that lies outside the representable range.

As a notational convenience, the bitwise operations are extended to operate elementwise on words as follows:

Definition 2.7 (Bitwise operations on words)

$$\begin{aligned}
\langle x_{w-1}\cdots x_0\rangle \wedge \langle y_{w-1}\cdots y_0\rangle &:= \langle x_{w-1}\wedge y_{w-1}\cdots x_0\wedge y_0\rangle \\
\langle x_{w-1}\cdots x_0\rangle \vee \langle y_{w-1}\cdots y_0\rangle &:= \langle x_{w-1}\vee y_{w-1}\cdots x_0\vee y_0\rangle \\
\langle x_{w-1}\cdots x_0\rangle \oplus \langle y_{w-1}\cdots y_0\rangle &:= \langle x_{w-1}\oplus y_{w-1}\cdots x_0\oplus y_0\rangle \\
\neg\langle x_{w-1}\cdots x_0\rangle &:= \langle \neg x_{w-1}\cdots \neg x_0\rangle
\end{aligned}$$

This is also the semantics bitwise operations have in programming languages such as C.

2.3 Bit Arithmetic

Bit vectors have no inherent meaning. Though they look like binary numbers when we write them down on paper, and we saw in section 2.1.2 how we can encode natural numbers as bit vectors, they do not innately "know" how to perform arithmetic operations such as addition. If we wish to perform an operation on a mathematical object represented as a bit vector, we must precisely specify that operation in terms of bit operations. Our goal is to specify the operation such that the resulting bit vector, when decoded, corresponds to the result we would have obtained if we operated directly on the mathematical objects. For a mathematical operation $\odot$, we will write $\odot^{\langle\rangle}$ for the equivalent operation on numbers encoded as bit vectors.

The algorithms for arithmetic we will introduce are largely similar to those you were hopefully taught for base-10 numbers as a child. A large part of learning how to perform binary arithmetic is to deconstruct what has long since become intuitive and look at the actual operations we implicitly perform when adding or multiplying numbers.

2.3.1 Addition

We wish to define an operation $+^{\langle\rangle}$ such that

$$\text{Bits2N}(\text{N2Bits}_w(y) +^{\langle\rangle} \text{N2Bits}_w(y)) = x + y \tag{2.18}$$

Adding binary numbers is much like adding decimal numbers. Starting from the least significant (rightmost) bits, we add them elementwise, keeping a carry. Example for adding $x +^{\langle\rangle} y = s$ where $x = \langle 01011\rangle, y = \langle 01001\rangle$:

[1] In fact, this encoding, known as *biased numbers* will make in appearance in chapter 3

i	x_i	y_i	s_i	c_i
0	1	1	0	1
1	1	0	0	1
2	0	0	1	0
3	1	1	0	1
4	0	0	1	0

The result is $s = \langle 10100 \rangle$ with no carry. In terms of bit operations, we can express the computation of sums and carries as follows (recall that $\oplus$ means exclusive-or):

$$s_0 = x_i \oplus y_i \tag{2.19}$$

$$c_0 = x_i \wedge y_i \tag{2.20}$$

$$s_i = x_i \oplus y_i \oplus c_{i-1} \tag{2.21}$$

$$c_i = (x_i \wedge y_i) \vee ((x_i \vee y_i) \wedge c_{i-1}) \tag{2.22}$$

Thus, the definition of $+^{\langle\rangle}$ is as follows, when adding two natural numbers represented as a w-bit word.

Definition 2.8 (Integer addition)

$$\langle x_{w-1} \cdots x_0 \rangle +^{\langle\rangle} \langle y_{w-1} \cdots y_0 \rangle := \langle s_{w-1} \cdots s_0 \rangle$$

where s_i, c_i are as in eqs. (2.19) *to* (2.22).

Our definition only covers the case where the two operands have the same number of bits. We can always *zero-extend* a w-bit word to a $l+w$-bit word by prepending l bits, without changing the natural number it encodes via Bits2N. On the other hand, *truncation* by removing bits can change the encoded value.

2.3.1.1 Overflow

Our definition of $+^{\langle\rangle}$ accepts and produces w-bit words. What happens if the resulting number is larger than $2^w - 1$ and thus logically requires more than w bits to be represented? Following definition 2.8, we see that the final carry bit c_{w-i} is not part of the result—if this bit is 1, then we say that the addition has *overflowed*.

Definition 2.9 (Integer overflow) *When the result of an integer operation is so large that it does fit in the designated word.*

With the integer representation we have used so far, the result of an overflow *wraps around* back to zero. This is conceptually similar to adding $5 + 5$ in decimal arithmetic and limiting the result to a single digit.

Example 2.3 ($\langle 11 \rangle +^{\langle\rangle} \langle 10 \rangle$)

$$s_0 = 1 \oplus 0 = 1 \tag{2.23}$$
$$c_0 = 1 \wedge 0 = 0 \tag{2.24}$$
$$s_1 = 1 \oplus 1 \oplus 0 = 0 \tag{2.25}$$
$$c_1 = (1 \wedge 1) \vee ((1 \vee 0) \wedge 0) = 1 \tag{2.26}$$

This gives us the final sum

$$\langle 11 \rangle +^{\langle\rangle} \langle 10 \rangle = \langle 01 \rangle \tag{2.27}$$

and since c_1 *is set, the computation has overflowed.*

Some programming languages make the last carry bit available as an *overflow bit* that programmers can check to see if overflow occurred. In others, an error is signalled if the overflow bit is set after an addition. But in many languages, such as C, overflow silently occurs and means the program can produce a possibly unexpected result when working with large numbers.

Does this mean that our carefully specified encoding of natural numbers as a fixed quantity of bits is simply mathematically wrong, when even something as simple as addition can give us an unexpected result? Not exactly: while no fixed-size encoding can encompass the infinite natural numbers, our encoding models the *ring of natural numbers modulo* 2^w. Our definition of arithmetic is *modular arithmetic*, which is mathematically quite well behaved. In particular, arithmetic has the expected algebraic properties (associativity, commutativity, 0 as additive identity, etc).

2.3.2 Bit Shifting

Given a w-bit word, we can *shift* the bits of the word left by k positions, inserting 0 bits in the newly vacated spots. Put another way, we discard the leftmost k bits and append k zeroes to the end.

Definition 2.10 (Logical left-shift by k bits)

$$\langle x_{w-1} \cdots x_0 \rangle << k := \langle x_{w-1-k} \cdots x_0 \underbrace{0 \cdots 0}_{k} \rangle$$

Left-shifting by k bits is equivalent to multiplying by 2^k when using the encoding of natural numbers from definition 2.4. This is analogous to multiplying a decimal integer by 10 by appending 0. Note that like addition, this is susceptible to overflow, as we discard the original k leftmost bits.

Example 2.4 (Left-shifting bit vectors with $w = 4$)

$$\mathrm{N2Bits}_4(3) << 2 = \langle 0011 \rangle << 2 = \langle 1100 \rangle = \mathrm{N2Bits}_4(12) \tag{2.28}$$
$$\mathrm{N2Bits}_4(9) << 2 = \langle 1001 \rangle << 2 = \langle 0100 \rangle = \mathrm{N2Bits}_4(8) \tag{2.29}$$

An obvious dual operation is *right-shifting*, where we drop the k leftmost bits and prepend k zero bits.

Definition 2.11 (Logical right-shift by k bits)

$$\langle x_{w-1} \cdots x_0 \rangle >> k := \langle \underbrace{0 \cdots 0}_{k} x_{w-1} \cdots x_k \rangle$$

Interpreted as an unsigned number, right-shifting a bit vector by k is equivalent to dividing by 2^k and then rounding towards zero. In the C programming languages, shifting is provided with the operators >> and <<, although with an important quirk that we will discuss in section 2.4.2.2.

Closely related to shifting is *rotation*, where bits are not discarded but merely moved to the other end of the word. We will not use bit rotation in these notes, but it is often efficiently supported by computers and has some niche uses.

2.3.3 Multiplication

In the section 2.3.2 we saw how left-shifting can be used to multiply by powers of two. General multiplication is a bit more involved, but can be done using essentially the same algorithm you learned in elementary school, only with bits instead of digits. More efficient algorithms exist, but are much more complicated. The formula for computing the product $z = x \times y$ is

$$z = \sum_{i=0}^{k-1} (x \times y_i) \times 2^i \tag{2.30}$$

where y_i is the ith bit of y. Note:

- The product $x \cdot y_i$ is multiplying a number with a single bit, meaning the result is either x or 0, which we can compute by using a logical-and on every bit of x.
- The multiplication with 2^i is with a power of 2, which can be done as in definition 2.10.

This lets us express the formula in terms of bit operations.

Definition 2.12 (Integer multiplication)

$$\langle x_{w-1} \cdots x_0 \rangle \times^{\langle\rangle} \langle y_{w-1} \cdots y_0 \rangle := \sum_{i=0}^{w-1} \langle x_{w-1} \wedge y_i, \ldots, x_0 \wedge y_i \rangle << i$$

where summation is with $+^{\langle\rangle}$.

Example 2.5 (5 × 3)

$$\begin{aligned}
\langle 0101\rangle \times^{\langle\rangle} \langle 0011\rangle &= \sum_{i=0}^{3} \langle 0 \wedge y_i, 1 \wedge y_i, 0 \wedge y_i, 1 \wedge y_i\rangle << i && (2.31)\\
&= \langle 0 \wedge 1, 1 \wedge 1, 0 \wedge 1, 1 \wedge 1\rangle << 0 && (2.32)\\
&\quad +^{\langle\rangle} \langle 0 \wedge 1, 1 \wedge 1, 0 \wedge 1, 1 \wedge 1\rangle << 1 \\
&\quad +^{\langle\rangle} \langle 0 \wedge 0, 1 \wedge 0, 0 \wedge 0, 1 \wedge 0\rangle << 2 \\
&\quad +^{\langle\rangle} \langle 0 \wedge 0, 1 \wedge 0, 0 \wedge 0, 1 \wedge 0\rangle << 3 \\
&= (\langle 0101\rangle << 0) && (2.33)\\
&\quad +^{\langle\rangle} (\langle 0101\rangle << 1) \\
&\quad +^{\langle\rangle} (\langle 0000\rangle << 2) \\
&\quad +^{\langle\rangle} (\langle 0000\rangle << 3) \\
&= \langle 0101\rangle +^{\langle\rangle} \langle 1010\rangle +^{\langle\rangle} \langle 0000\rangle +^{\langle\rangle} \langle 0000\rangle && (2.34)\\
&= \langle 1111\rangle && (2.35)
\end{aligned}$$

2.4 Signed Numbers

The number representation discussed thus far can only represent non-negative numbers, which in computer science jargon are called *unsigned*. If we want to handle negative numbers as well, we need a *signed* representation.

2.4.1 Sign-magnitude

In normal number notation, we turn a number negative by prefixing it with a minus sign. Thus, an obvious way to introduce negative numbers is to treat the most significant (leftmost) bit as a *sign bit*, which is set when the number is negative. This representation is known as *sign-magnitude*. The conversion function is as follows:

Definition 2.13 (Bit vector to integer using sign-magnitude)

$$\text{SM2Int}(\langle x_{w-1} \cdots x_0\rangle) := -1x_{w-1} \times \text{Bits2N}(\langle x_{w-2} \cdots x_0\rangle)$$

Arithmetically negating a number in sign-magnitude representation is quite simple: just negate the sign bit.

Although simple, this representation has the downside that it contains two representations of zero, as seen in table 2.1. While having multiple representations of the same value is not inherently wrong, it is slightly wasteful, and complicates the definition of arithmetic. In particular, arithmetic and comparisons of sign-magnitude numbers is somewhat involved because we have to treat the sign bit specially. For these reasons, sign-magnitude is not used for integers in most modern computers.

x	SM2Int(x)	x	SM2Int(x)
$\langle 0000\rangle$	0	$\langle 1000\rangle$	0
$\langle 0001\rangle$	1	$\langle 1001\rangle$	-1
$\langle 0010\rangle$	2	$\langle 1010\rangle$	-2
$\langle 0011\rangle$	3	$\langle 1011\rangle$	-3
$\langle 0100\rangle$	4	$\langle 1100\rangle$	-4
$\langle 0101\rangle$	5	$\langle 1101\rangle$	-5
$\langle 0110\rangle$	6	$\langle 1110\rangle$	-6
$\langle 0111\rangle$	7	$\langle 1111\rangle$	-7

Table 2.1: All possible four-bit words interpreted as integers using sign-magnitude representation.

2.4.2 Two's complement

The overwhelmingly most common integer representation in modern computers is *Two's Complement*. In this representation, a negative number is encoded by the logically negated bit sequence of the corresponding unsigned positive number, plus one. This lets us define an encoding function.

Definition 2.14 (Integer to Two's Complement)

$$\text{Z2TC}_w(x) = \begin{cases} \text{N2Bits}_w(x) & x \geq 0 \\ \neg\text{N2Bits}_w(|x|) +^{\langle\rangle} \text{N2Bits}_w(1) & x < 0 \end{cases}$$

Note that we are using word negation (definition 2.7) in the above formula.

An equivalent view of Two's Complement is that it represents integers by assigning each bit a weight, just like with unsigned numbers, but assigns the most significant bit (the *sign bit*) a large *negative* weight. This is the intuition we use in our decoding function.

Definition 2.15 (Two's Complement to Integer)

$$\text{TC2Z}(\langle x_{w-1} \cdots x_0\rangle) := -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Using Two's Complement, all distinct bit vectors represent distinct integers, as demonstrated on table 2.2. But this representation also has other useful properties. One almost miraculous property is that we can add and multiply numbers in Two's Complement representation the exact same way that we add unsigned numbers using definition 2.8, and get the right result (ignoring overflow). For non-negative numbers, this is not terribly surprising, as these

x	TC2Int(x)	x	TC2Int(x)
$\langle 0000 \rangle$	0	$\langle 1000 \rangle$	-8
$\langle 0001 \rangle$	1	$\langle 1001 \rangle$	-7
$\langle 0010 \rangle$	2	$\langle 1010 \rangle$	-6
$\langle 0011 \rangle$	3	$\langle 1011 \rangle$	-5
$\langle 0100 \rangle$	4	$\langle 1100 \rangle$	-4
$\langle 0101 \rangle$	5	$\langle 1101 \rangle$	-3
$\langle 0110 \rangle$	6	$\langle 1110 \rangle$	-2
$\langle 0111 \rangle$	7	$\langle 1111 \rangle$	-1

Table 2.2: All possible four-bit words interpreted as integers using Two's Complement representation.

have identical representations in unsigned and Two's Complement representation. For negative numbers, the reason this works is that negative numbers in Two's Complement maintain their relative ordering when interpreted as unsigned numbers, which is not the case for sign-magnitude. It is likely that it is this property that has made Two's Complement the dominant integer representation, as it means a computer designer can use the same circuitry for computing signed and unsigned numbers. Only when relatively ordering Two's Complement numbers does a computer need to inspect the sign bit.

2.4.2.1 Arithmetic Negation

Arithmetic negation of Two's Complement numbers is done by logically negating all the bits and then incrementing by one:

Definition 2.16 (Negating a Two's Complement number)

$$\mathrm{negTC}(\langle x_{w-1} \cdots x_0 \rangle) := \langle \neg x_{w-1} \cdots \neg x_0 \rangle +^{\langle\rangle} \langle 0 \cdots 1 \rangle$$

However, because the range of Two's Complement is asymmetric—there are more negative than positive numbers—negation of the most negative number is an identity operation.

Example 2.6 (Overflow when negating)

$$\begin{aligned}
\mathrm{negTC}(\langle 1000 \rangle) &= \langle \neg 1 \neg 0 \neg 0 \neg 0 \rangle +^{\langle\rangle} \langle 0001 \rangle &(2.36)\\
&= \langle 0111 \rangle +^{\langle\rangle} \langle 0001 \rangle &(2.37)\\
&= \langle 1000 \rangle &(2.38)
\end{aligned}$$

Nevertheless, the definition of negation is sufficient for us to define subtraction of numbers in Two's Complement:

Definition 2.17 (Integer subtraction in Two's Complement)

$$x \mathbin{-^{\langle\rangle}} y := x +^{\langle\rangle} \text{negTC}(y)$$

2.4.2.2 Arithmetic right shift

In section 2.3.2 we saw how shifting by k corresponds to multiplication and division by 2^k. But while left-shifting works equivalently for unsigned and Two's Complement, right shifting a negative number tends not to produce the arithmetically correct result.

Example 2.7 (Logically right-shifting Two's Complement numbers)

$$\text{Z2TC}(4) >> 1 \quad = \langle 0100 \rangle >> 1 \quad = \langle 0010 \rangle \quad = \text{Z2TC}(2) \tag{2.39}$$
$$\text{Z2TC}(-4) >> 1 \quad = \langle 1100 \rangle >> 1 \quad = \langle 0110 \rangle \quad = \text{Z2TC}(6) \tag{2.40}$$

We can consider an unsigned number as consisting of an arbitrary number of 0 bits to the left of its most significant bit, in the same manner we can write decimal numbers with as many leading zeroes as we desire. When we right-shift, it is these zeroes that are inserted. Following this analogy, a negative Two's Complement number could be viewed as having an arbitrary number of 1 bits to the left of it—or more generally, copies of the sign bit. When we right-shift, it is *these* that should be inserted.

To address this issue, we define a new kind of right-shift where we do not prepend zeroes, but instead copies of the sign bit. This is called an *arithmetic right shift*.

Definition 2.18 (Arithmetic right-shift by k bits)

$$\langle x_{w-1} \cdots x_0 \rangle >>^{\text{a}} k := \langle \underbrace{x_{w-1} \cdots x_{w-1}}_{k} x_{w-1} \cdots x_k \rangle$$

Example 2.8 *Arithmetically right-shifting negative numbers*

$$\text{Z2TC}(4) >>^{\text{a}} 1 \quad = \langle 0100 \rangle >>^{\text{a}} 1 \quad = \langle 0010 \rangle \quad = \text{Z2TC}(2) \tag{2.41}$$
$$\text{Z2TC}(-4) >>^{\text{a}} 1 \quad = \langle 1100 \rangle >>^{\text{a}} 1 \quad = \langle 1110 \rangle \quad = \text{Z2TC}(-2) \tag{2.42}$$

There is no need to define an arithmetic *left* shift, as left-shifting already behaves as desired. In programming languages that distinguish between signed and unsigned numbers in their type system, such as C, right-shifting a signed number performs an arithmetic shift, and right-shifting an unsigned number performs a logical shift.

2.4.3 Sign Extension

In section 2.3.1 we saw that a w-bit unsigned number can be extended to a $w + k$-bit number by prepending zeroes. Such *zero extension* can however change the numeric interpretation of a Two's Complement number, for reasons similar to the troubles we had with right-shifting.

Example 2.9 (Zero extension may change value)

$$\text{TC2Int}(\langle 1100 \rangle) = -4 \tag{2.43}$$
$$\text{TC2Int}(\langle 00001100 \rangle) = 12 \tag{2.44}$$
$$\text{TC2Int}(\langle 0100 \rangle) = 4 \tag{2.45}$$
$$\text{TC2Int}(\langle 00000100 \rangle) = 4 \tag{2.46}$$

To extend a w-bit word to a $w-k$-bit word while preserving its Two's Complement numeric value, we perform *sign extension* where we prepend copies of the sign bit.

Example 2.10 (Sign extension preserves value)

$$\text{TC2Int}(\langle 1100 \rangle) = -4 \tag{2.47}$$
$$\text{TC2Int}(\langle 11111100 \rangle) = -4 \tag{2.48}$$
$$\text{TC2Int}(\langle 0100 \rangle) = 4 \tag{2.49}$$
$$\text{TC2Int}(\langle 00000100 \rangle) = 4 \tag{2.50}$$

Chapter 3

Floating-Point Numbers

Integers are all well and good, but we often need to solve problems that require the use of fractions—that is, we need a machine representation of some kind of approximation of rational or real numbers. In this section we will look at the various solutions one can come up with to this problem, but ultimately we will focus in detail on *floating-point* numbers, which are the most common implementation of fractional numbers in modern computers. This is an *enormous* topic, and many devote their entire careers to studying number representations. This chapter is relatively superficial, but the particularly interested student can continue their studies by reading the *Handbook of Floating-Point Arithmetic* [5], from which most of the information in this chapter is sourced, or taking courses on numerical analysis.

3.1 The Basic Problem

Fractional numbers are inherently more difficult to implement on computers because of their density. Between the numbers x and y, there are $|x - y|$ integers. Thus, when we define a fixed-size integer representation, there will be a smallest and largest number, but there will be no gaps inside this interval. In contrast, between x and y there is an *infinite* number of rational numbers (assuming $x \neq y$). With a w-bit fixed-size representation we can at most represent 2^w distinct values, so any nonempty interval of rational numbers will necessarily have numbers we cannot represent using a fixed-size encoding. Number representations using an arbitrary and value-dependent number of bits can be defined that can represent any computable real numbers using a variable number of bits, but these are implemented in software rather than in hardware, and are *much* slower than fixed-size representations.

One obvious representation is to represent fractions as pairs of integers. For example, the fraction

$$\frac{1}{3}$$

can be straightforwardly represented as the pair $(1, 3)$. Using w-bit integers,

such a pair could be represented as a $2w$-bit vector. The main problem with this representation is that a given number has many possible encodings—e.g. $(1, 3)$ and $(2, 6)$ both represent the same number. Fractions can be reduced using algorithms that have been known for literally thousands of years, but these are computationally expensive—certainly not something we want a computer to do as part of routine arithmetic. Another consequence of this redundancy is that we also waste encoding space. As much as possible, we desire a representation where distinct bit patterns encode distinct numbers.

3.1.1 Precision and Accuracy

Informally, the terms *precision* and *accuracy* are often used interchangeably. In the following we will use them with very specific meanings.

Precision is how many digits are present in a result. As an analogy, if someone asks you for the current time and you answer "twelve hours, four minutes, three seconds, fourteen miliseconds", then your answer is very precise.

Accuracy is how close an approximated or rounded result is to the *true* value. Following on the analogy above, that very precise answer may be quite inaccurate if it's actually 8 in the morning. Accuracy is also sometimes called *exactness*.

Since any finite-sized representation of rational numbers will always have gaps, the result of any computation must be rounded to the nearest number that can actually be represented in limited precision. This is a source of inaccuracy. While we focus mostly on representation rather than calculation, we will return to the issue of rounding in section 3.5.

3.2 Fixed-point Fractional Numbers

In previous sections we saw that that when we write a binary integer such as 10010101_2 it is basically interpreted the same way as 149_{10}—the structure is the same, except that the digit weights are powers of 2 instead of powers of 10. This correspondence inspired the unsigned integer representation of definition 2.4. Can we perhaps use the same idea to represent fractional numbers? Yes. A decimal fraction 123.456_{10} can be viewed as having weights that are *non-negative* powers of 10 on the left of the decimal point, and *negative* powers to the right. To compute its value, we multiply each digit with the weight.

Example 3.1 (Interpretation of decimal fraction 123.456_{10})

Digit	1		2		3	.	4		5		6
Weight	10^2		10^1		10^0		10^{-1}		10^{-2}		10^{-3}
Sum	100	+	20	+	3	+	$\frac{4}{10}$	+	$\frac{5}{100}$	+	$\frac{6}{1000}$

Similarly, a binary fractional number 1001.0101 has weights that are powers of 2, and the ones to the right of the *binary point*[1] have negative exponents:

Example 3.2 (Interpretation of binary fraction 1001.0101_2)

Digit	1		0		0		1	.	0		1		0		1
Weight	2^3		2^2		2^1		2^0		2^{-1}		2^{-2}		2^{-3}		2^{-4}
Sum	8	+	0	+	0	+	1	+	0	+	$\frac{1}{4}$	+	0	+	$\frac{1}{16}$
	$= 9.3125_{10}$														

For some word size w, we must specify how many of the bits we allocate to the integral part (before the binary point) and how many are part of the fraction. In the example above we used a symmetric representation with 4 bits devoted to the integral and fractional part. The crucial property is that this allocation is fixed for a given format, and cannot vary. This is why this number representation is called *fixed point*.

Definition 3.1 (Fixed-point number) *A number representation where a fixed number of bits is allocated to the fractional part.*

Definition 3.2 (Bit vector interpreted as unsigned fixed point)

$$\text{Fix2Q}(\langle x_{n+m-1} \cdots x_m \; x_{m-1} \cdots x_0 \rangle) = \sum_{i=0}^{n+m-1} x \cdot 2^{i-m}$$

where n, m is the number of bits allocated to integral and fractional part, respectively.

For simplicity we deal only with non-negative numbers in this representation. Support for negative numbers can be implemented by either adding a sign bit, or by interpreting the integral part as a Two's Complement number.

One nice property of fixed-point representation is that we can divide and multiply by bit-shifting, just as with integers. Unfortunately, it also has serious limitations.

Limitation #1 This representation can only represent numbers of the form

$$\frac{x}{2^k}$$

for some x, k. This means that numbers such as 0.3_{10} cannot be represented exactly, but must be approximated. Allowing 4 bits for the fraction, the best approximation is

$$0.0101_2 = 0.3125_{10}$$

This is not a restriction that is particular to our use of binary. Any radix will have numbers that cannot be represented, just as we cannot write

$$\frac{1}{3}$$

with any finite decimal sequence.

[1] Corresponding to the *decimal point* in a base-10 system.

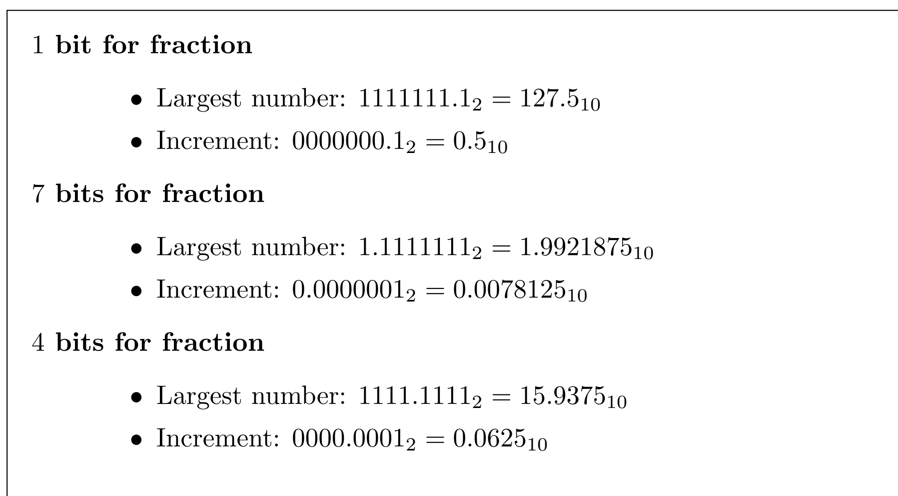


Figure 3.1: The fixed-point dilemma for $w = 8$. The *increment* is the distance between neighbouring numbers.

Limitation #2 Given w bits to represent fixed-point numbers, we need to decide once and for all how many bits we dedicate to the integral part, and how many to the fraction. If we allocate many bits to the integral part, we will be able to represent larger numbers, but the distance between neighbouring representable numbers will be breater. This is illustrated on fig. 3.1.

Note in particular that fixed-point representations have lower relative precision for numbers close to zero. Suppose $w = 8$ and we are using 4 bits for the fraction; then the next highest number from 1 is 1.00625—a relative distance of 0.00625. But for 15, the next number is 15.00625—a relative distance of 0.00004167.

Ideally, we want a number representation where the relative distance between numbers is constant—meaning that the absolute distance between representable numbers close to zero is small, while the distance between numbers far from zero is large. Another way of looking at this of the 2^w distinct numbers possible for a w-bit representation, most of them should be close to zero. We can view this as a number comprising w digits where the binary point is not fixed, but rather "floating", which is why the common name for this type of numbers is *floating-point numbers*.

3.3 Intuition behind floating-point numbers

Before delving into the technical details of floating-point numbers, let us build up a little intuition for how they work. floating-point numbers (or just *floats*) are inspired by decimal scientific notation where a number is represented as

$$x = \alpha \times 10^{\beta} \tag{3.1}$$

where the *significand* α is any number and the exponent *exponent* β is an integer. Obviously we can represent any number simply by setting $\beta = 0$ and using only α. However, a common convention is to require that $|\alpha| < 10$, in which case we say a number is *normalised*.

Example 3.3 (Normalisation in scientific notation) *The number*

$$x = -123 \times 10^{-1}$$

is not normalised, but we can normalise it by dividing the significand by 10 *and incrementing the exponent by* 1*, giving*

$$x - 1.23 \times 10^{1}.$$

Now suppose we make this representation finite by requiring

$$-100 < \beta < 100$$

and also that α has exactly two digits to the right of the decimal point, meaning it is always of the form $x.xx$, possibly with a sign. This means we can now encode any number as three digits for α (and a sign) plus two digits for β (and a sign). But what are the consequences for which numbers can be represented?

One consequence is that we now have a largest representable number, namely the one where $\alpha = 9.99$ and $\beta = 99$. Another is that the distance between neighbouring numbers now depends on the exponent β.

Example 3.4 (Distances between neighbouring numbers)

$$|1.23 \times 10^{1} - 1.24 \times 10^{1}| = 0.1 \tag{3.2}$$

$$|9.99 \times 10^{1} - 0.01 \times 10^{2}| = 0.1 \tag{3.3}$$

$$|0.01 \times 10^{1} - 0.02 \times 10^{2}| = 0.2 \tag{3.4}$$

$$|1.23 \times 10^{50} - 1.24 \times 10^{50}| = 5 \tag{3.5}$$

This is the basic idea behind floating-point numbers, and while floating-point number formats have significant extra complexities to deal with various numerical challenges and to make them efficiently implementable on binary electronic computers, it all comes back to this idea of having a normalised scientific representation comprising a significand and exponent.

3.4 IEEE 754

Floating-point formats were widely supported even by the earliest computers, but they diverged widely in how they were implemented, how much precision they offered, and what semantics they used for rounding. This made it difficult for numerical programs written on one computer to run correctly on another computer. Significant effort was spent on standardising floating-point arithmetic, and in 1985, the IEEE 754 standard was ratified. It has since been

Name	binary16	binary32	binary64
Informal name	Half precision	Single precision	Double precision
C type	N/A	`float`	`double`
p	11	24	53
$e_{\max}$	+15	+127	+1023
$e_{\min}$	−14	−126	−1022

Table 3.1: The most common IEEE 754 floating-point formats. Half precision floats are not supported in standard C, but are relatively common in graphics programs. More exotic non-standard formats are also used in machine learning.

updated and extended, but the core concepts and principles are unchanged. The IEEE 754 standard is now essentially universally supported, except for very specialised processors.

Apart from specifying rules and encodings of conventional rational numbers, IEEE 754 also mandates the existence of certain special values:

- NaN ("not a number"), a family of special values that are produced by invalid operations such as $\sqrt{-1}$. The purpose of NaN is to ensure that all arithmetic operations are "closed", in that they produce a value that is representable in the IEEE 754 value representation, even though it is literally speaking *not a number*.
- Positive and negative infinity, which have two important uses. One is to represent the result of operations such as division by zero. Another is to represent overflow, where the result of an arithmetic operation lies outside the representable number range. Compare this to integers, where the most typical result is wraparound.

IEEE defines various floating-point formats. All of them are characterised by four integers

- a *radix*, which we will assume to be 2 although IEEE 754 also specifies decimal floating-point formats;
- a *precision* $p \geq 2$, roughly corresponding to the number of "significant bits" in the significand;
- two *extremal exponents* $e_{\min}, e_{\max}$ such that $e_{\min} < e_{\max}$. In all formats specified by IEEE 754, $e_{\min} = 1 - e_{\max}$.

The two most common IEEE 754 formats are binary32 and binary64, corresponding to the `float` and `double` types in most C-like programming languages. The parameters for the most common binary IEEE 754 formats are shown in table 3.1.

A finite floating-point number expressed in such a format is a number x for which there exists a representation (s, m, e) defined as follows.

Definition 3.3 (Representation of floating-point number x)

$$x = -1^s \cdot m \cdot 2^e$$

where

- s *is the* sign bit*, which is 1 for negative numbers;*
- m *is a number called the* normal significand*.*
- e *is an integer such that* $e_{min} \leq e \leq e_{max}$*, called the* exponent*.*

The possible values of m is related to p in a way that we will specify in section 3.4.1. Definition 3.3 does not specify unique representation for every number. Consider

$$16 = -1^0 \cdot 16 \cdot 2^0 = -1^0 \cdot 32 \cdot 2^{-1} \tag{3.6}$$

In order to ensure a unique representation for a given number, we require that floating-point numbers are normalised, by always choosing the representation for which the exponent is smallest (but of course not less than $e_{\min}$). This implies that $0 \leq m < 2$.

Further, whenever $1 \leq m < 2$, we say that x is a *normal* number. Intuitively we can see m as indicating a number of the form $1.xxx_2$, where m constitutes the binary digits to the right of the binary point. These digits are sometimes called the *fraction*.

Otherwise, when $e = e_{\min}$, we must also have that $0 \leq m < 1$, and we call this case a *subnormal* number. For subnormal numbers, the bit before the binary point must necessarily be 0. That is, in this case m indicates a number of the form $0.xxx_2$. This means that when we encode m in a bit vector, we can elide the bit before the binary point, because it can be inferred from e, thus saving a a bit.

The special case of zero will be discussed later.

3.4.1 Bit Vector Representation

By definition 3.3, a floating-point number is represented by three numbers s, e, m. As a bit vector, these are encoded as tree fields S, E and T. Further, some bit patterns are reserved to express NaN, infinity, and zero.

Definition 3.4 (Fields of a floating-point number x)

$$\langle \underbrace{S_0}_{1\ bit}\ \underbrace{E_{r-1} \cdots E_0}_{r\ bits}\ \underbrace{T_{p-2} \cdots T_0}_{p-1\ bits} \rangle$$

where S *encodes* s *directly,* E *encodes* e*, and* T *encodes* m*.*

The S field is one bit in length and directly corresponds to s. The E field consists of r bits and the T field consists of $p-1$ bits, but still represents p bits of information, because an extra bit can be deduced from the context as discussed above. How these fields are decoded as e and m depends on their exact values. In the normal case, E encodes e as a *biased* number, which we briefly saw in section 2.2.2. Specifically, E encodes an unsigned integer from which we subtract a *bias* b to obtain the possibly negative e. For all IEEE formats, $b = e_{\max}$, which lets E encode integers ranging from $e_{\min}$ to $e_{\max}$.

In total, decoding a floating point number represented as a bit vector in the form from definition 3.4 involves the following cases.

- If $E = \langle 1 \cdots 1 \rangle$ (a sequence of ones) and $T \neq \langle 0 \cdots 0 \rangle$, then the bit vector represents not-a-number (NaN).[2]
- If $E = \langle 1 \cdots 1 \rangle$ and $T = \langle 0 \cdots 0 \rangle$ then the bit vector represents $\pm\infty$, with sign determined by S.
- If $E = \langle 0 \cdots 0 \rangle$ and $T = \langle 0 \cdots 0 \rangle$, then the bit vector represents ± 0, with sign determined by S.
- If $E \neq \langle 0 \cdots 0 \rangle$ and $E \neq \langle 1 \cdots 1 \rangle$, then the bit vector represents the normal number
$$(-1)^s \cdot m \cdot 2^e$$
where
$$\begin{aligned} b &= e_{\max} \\ s &= \text{Bits2N}(S) \\ m &= 1 + \text{Bits2N}(T) \cdot 2^{1-p} \\ e &= \text{Bits2N}(E) - b \end{aligned}$$
- If $E = \langle 0 \cdots 0 \rangle$ and $T \neq \langle 1 \cdots 1 \rangle$, then the bit vector represents the subnormal number
$$(-1)^s \cdot m \cdot 2^{e_{\min}}$$
where
$$\begin{aligned} s &= \text{Bits2N}(S) \\ m &= \text{Bits2N}(T) \cdot 2^{1-p} \end{aligned}$$

These cases form a conversion function. For space reasons, we will not write it down in that form.

Example 3.5 (Interpreting a bit vector as a float) *Using the binary16 format from table 3.1, we interpret*

$$\langle 111010001010001 \rangle$$

[2] Note that *multiple* NaNs exist, since T can have multiple possible values. This is sometimes called the NaN *payload*, and can be used to store information about the origin of the NaN value.

as a float. First we split the bit vector into fields.

$$S = \langle 1 \rangle \quad E = \langle 11010 \rangle \quad T = \langle 0001010001 \rangle$$

Inspecing E and T, we see we are dealing with a normal number. We then compute

$$\begin{aligned} b &= 15 \\ s &= 1 \\ m &= 1 + 81 \cdot 2^{-10} \\ e &= 11 \end{aligned}$$

giving the final result

$$(-1)^s \cdot m \cdot 2^e = -2210$$

3.4.2 Interesting Properties

Working with floating-point numbers is a big topic, but there's a handful of simple properties that are useful to remember.

All floating point numbers can be arithmetically negated (multiplied by negative one), simply by negating the sign bit. This is in contrast to Two's Complement integers, where the most negative number cannot be negated.

In all IEEE floating-point formats, the range of e is symmetric around zero, e.g. table 3.1 shows that for binary32 ("single precision"), e ranges from -126 to $+127$. When $e < 0$, the magnitude of the represented number is less than 1. This means that approximately *half* of all representable numbers are in the interval $[-1, 1]$! Therefore, while we can represent astronomically large numbers with a magnitude of more than 2^{127}, only a relatively tiny portion of the encoding space is devoted to these, and any arithmetic in this end of the number line is likely going to be subject to severe rounding error.

3.5 Rounding

Most rational numbers have no representation as floating point numbers. Therefore, in general the result of an arithmetic operation cannot be represented exactly in any floating point format, but has to be *rounded* to the nearest representable value. As an example, although both 1 and 10 can be represented, the division

$$\frac{1}{10}$$

cannot be exactly represented in any binary floating point format, and instead has to be approximated. In binary32, the closest approximation is

$$13421773 \cdot 2^{-27} = 0.100000001490116 12.$$

In the first floating-point systems, the way results were rounded was not always specified, and frequently varied among computers. The IEEE 754 standard mandates the concept of *correct rounding*, where the result of a single

operation is calculated with infinite precision, and then rounded to a representable value using one of several possible *rounding modes*[3].

Definition 3.5 (Correct rounding) *When the computed result of an operation is the same as if the operation was done with infinite precision and unlimited range, then rounded according to the chosen rounding mode.*

Essentially, when an operation is correctly rounded, it means that the result is as close to being mathematically correct as we can represent within the floating-point format. IEEE 754 guarantees correct rounding for addition, subtraction, multiplication, division, and square root, but not for the various transcendental functions: logarithms, sines, cosines, etc. The exactness of these functions can vary between platforms.

3.5.1 Rounding modes

IEEE 754 specifies five rounding modes:

- Round towards $-\infty$, also called "rounding downwards". When rounding a number x, this produces the largest representable number smaller or equal to x.
- Round towards ∞, also called "rounding upwards". When rounding a number x, this produces the smallest representable number greater or equal to x.
- Round towards 0, which rounds downwards when $x > 0$ and otherwise rounds upwards.
- Round to nearest, ties to even. This rounds to the closest representable number, picking the even number in case of ties. This is the default rounding mode in the vast majority of systems.
- Round to nearest, ties away from zero. Picks the representable number with the largest magnitude in case of ties.

In most cases we do not worry about rounding modes when programming. The default behaves sensibly in most cases, and changing the rounding mode is usually a global change for the entire process, which makes it difficult to use in a modular way.

[3] The 2008 revision to IEEE 754 changed the term to *rounding direction attribute*, but *rounding mode* is still in widest use by practitioners.

3.6 Arithmetic

While a full discussion on implementing floating-point arithmetic is well outside the scope of this text, we are going to briefly discuss how multiplication and addition is implemented. For clarity of exposition, we are going to assume that the operands are proper numbers expressed as

$$(-1)^s \cdot m \cdot 2^e.$$

In particular, they are not infinities or NaN (but may be subnormal). In general, when an operation has a NaN operand, the result is also a NaN. Infinities are produced by overflow, and most operations likewise preserve infinity, e.g:

$$1 + \infty = \infty \tag{3.7}$$

But some operations involving infinity may also produce NaN:

$$\infty - \infty = \text{NaN} \tag{3.8}$$

Definition 3.6 (Floating-point overflow) *When the magnitude of a floating-point number becomes too large to be representable and the result becomes* $\pm\infty$*.*

Definition 3.7 (Floating-point underflow) *When the magnitude of a floating-point number becomes too small be representable and the result becomes* 0*.*

Note that overflow also occurs when a number becomes *too negative*. This is in contrast to integers, where this situation is called *integer underflow*. With floating-point numbers, underflow means that the number comes too close to zero, and ends up being rounded to zero.

3.6.1 Multiplication

Due to the exponential representation, multiplication of floating-point numbers is actually simpler than addition. The product of two floating-point numbers

$$(-1)^{s_1} \cdot m_1 \cdot 2^{e_1}$$

and

$$(-1)^{s_2} \cdot m_2 \cdot 2^{e_2}$$

is given by

$$(-1)^{s_3} \cdot m_3 \cdot 2^{e_3} = ((-1)^{s_1} \cdot m_1 \cdot 2^{e_1}) \cdot ((-1)^{s_2} \cdot m_2 \cdot 2^{e_2}) \tag{3.9}$$

where

$$s_3 = s_1 \oplus s_2 \tag{3.10}$$

$$m_3 = m_1 \cdot m_2 \tag{3.11}$$

$$e_3 = e_1 + e_2 \tag{3.12}$$

This might produce an un-normalised or unrepresentable number. If $m_e \geq 2$, meaning that the number is not normalised, we shift m_3 right (equivalent to repeatedly dividing by two) and increment e_3 (equivalent to doubling). Then, if e_3 is out of range (either less than $e_{\min}$ or greater than $e_{\max}$), the operation has overflowed, producing $\pm\infty$. Finally, we round m_3 to p bits. The addition of the exponents is just integer addition, which is straightforward. Implementing the multiplication of the significands is more complicated, but fortunately outside the scope of this text.

3.6.2 Addition

Given two floating-point numbers

$$x_1 = (-1)^{s_1} \cdot m_1 \cdot 2^{e_1}$$

and

$$x_2 = (-1)^{s_2} \cdot m_2 \cdot 2^{e_2}$$

we seek to compute the sum

$$((-1)^{s_3} \cdot m_3 \cdot 2^{e_3}) = x_1 + x_2 \tag{3.13}$$

meaning we have to find s_3, m_3, e_3.

First, assume without loss of generality that $e_1 \geq e_2$[4]. We rewrite x_2 such that its exponent becomes e_1, but without changing its numerical value by also constructing a new significand m_2':

$$x_2' = ((-1)^{s_2} \cdot m_2' \cdot 2^{e_1} \tag{3.14}$$

Note that x_2' is most likely not a normal representation, but this is fine, as it is merely an intermediate result. Now we can compute

$$s_3 = s_1 \oplus s_2 \tag{3.15}$$

$$m_3 = m_1 + m_2' \tag{3.16}$$

As with multiplication, this might not yield a normalised or representable number. We fix it a similar way. If $m_3 \geq 2$, we shift m_3 right and increment e_3. If $m_3 < 1$, we shift m left and decrement e_3. If e_3 is outside the range $[e_{\min}, e_{\max}]$, the result is $\pm\infty$. Otherwise, we round m_3 to p bits.

Example 3.6 (Addition with $p = 3$)

$$\begin{array}{lll}
 & (-1.01 \cdot 2^2) + (1.1 \cdot 2^4) & \\
= & (-1.01 \cdot 2^2) + (110.0 \cdot 2^2) & \textit{Align exponents} \\
= & (-1.01 + 110.0) \cdot 2^2 & \textit{Distributivity} \\
= & 100.11 \cdot 2^2 & \textit{Add significands} \\
= & 1.0011 \cdot 2^4 & \textit{Normalise} \\
= & 1.01 \cdot 2^4 & \textit{Perform rounding}
\end{array}$$

[4]We can always just flip the operands to the addition.

Chapter 4

Data as Bytes

In the previous chapters we looked at data represented as bits. While fiddling with bits is indeed how computation is ultimately carried out in a physical sense, computers are built as layers of aggregation and abstraction. We already saw how individual bits were assembled into bit vectors, and how bit vectors of known lengths can be used to represent data types such as numbers. In principle, we could imagine a computer that simply treated the totality of available data as a single large bit vector. In practice, the smallest storable unit of data is a bit vector called a *byte*, which is stored in *memory*, and which is the subject of this chapter.

The size of a used to vary between computers, and tended to be the amount of bits necessary to represent a single character. Many older computers used 6-bit bytes, as $2^6 = 64$ allows enough distinct values to represent the alphabet, ten digits, and some punctuation characters[1]. However, essentially all modern computers use 8-bit bytes. In fact, "byte" has become more or less synonymous with "8 bits", although a pedant would insist that the proper term for such a bit vector is *octet*. For simplicity, we will use "byte" in this text.

As we saw earlier, many data types are represented in more than 8 bits. To represent such data, we need multiple bytes. Because the byte unit is the unit of storage, all common data types are a multiple of 8 bits, meaning one or more whole bytes. Working with data that is not expressible as a whole number of bytes is possible, but is more of a chore.

When discussing values at the byte level, hexadecimal (base-16) notation is often used, where the letters *a*–*f* (or their uppercase equivalents) are used to denote digits with values 10–15. Hexadecimal notation is convenient because the 16 different digits for a single hexadecimal digit corresponds exactly to 4 bits, and a single 8-bit byte can thus be written using exactly two hexadecimal digits.

$$149_{10} = 10010101_2 = 95_{16} \tag{4.1}$$

It is usually easier to read a hexadecimal number than a long binary number.

[1] We will return to the issue of text representation in section 4.3

Figure 4.1: Memory is an array of bytes, indexed from 0 to some largest address, depending on the size of the memory. Addresses are typically written with hexadecimal (base-16) numbers.

Also, it is easy to translate hexadecimal numbers to binary, because we can translate each digit separately and simply concatenate at the end.

4.1 Memory

The main working storage of a computer is called the *memory*. Conceptually, the memory can be seen as a large contiguous array of bytes. Given an index, called an *address*, we can retrieve or modify the byte stored at that address. Figure 4.1 shows this conceptual model. This kind of memory is called *random access memory* (or RAM) because we can at any time access the data at any address. When a program is running, its data (values of variables and objects) is stored in memory.

Physically, memory is usually implemented using *dynamic RAM* (DRAM) technology, although other technologies have been used in the past. DRAM needs to be constantly refreshed, requiring power, or it will lose data. This means that data stored in DRAM will be lost when the computer is turned off. Data that we intend to keep must be written to persistent storage, which today typically takes the form of a hard disk drive (HDD) or solid state drive (SSD), although many other technologies exist. These storage technologies are not typically considered part of "memory", but are made available through *file systems*, although the line is somewhat blurred by the notion of *virtual memory* where data is transparently moved between different physical storage media (outside the scope of this chapter).

In principle, to perform a computation such as addition, the processor must fetch the operands from memory, perform the operation as described in previous chapters, and write the result back to memory. In practice, processors contain a fixed set of *registers* that we can view as a small collection of fixed-size variables. The processor operates on data stored in registers, and copies explicitly between registers and memory as needed. As programmers we can usually ignore this distinction—it is dealt with by the implementation of the programming language we are using. We will return to this in chapter 5.

4.2 Multi-byte Words

When we store a multi-byte object in memory, such as a 64-bit floating-point number, we identify it with the address of its *first* constituent byte, counting from low addresses to high addresses. As always, data is not self-describing, so it is in principle our responsibility to remember the number of constituent

bytes. For simple data types, the size is known. For example, in C on a modern x86-64 machine, the `char` type is always 1 byte, a `short` is 2 bytes, and an `int` is 4 bytes. We will discuss variable-size data types in section 4.3 and chapter 6.

4.2.1 Byte Ordering

When storing a multi-byte object (say, a 4-byte `int`) in memory, each individual byte will have an address. This means that the *byte order* is visible at the memory level, and hence that the order in which we store the bytes matter. This is called *endianness*, a term taken from *Gulliver's Travels* where the Lilliputians fight a civil war over whether the shell of a boiled egg should be broken from the big or little end.

In computer science, there are two main endianness conventions:

Definition 4.1 (Big Endian) *The most significant byte is stored at the lowest address.*

Definition 4.2 (Little Endian) *The least significant byte is stored at the lowest address.*

Example 4.1 (Representing $x = 01234567_{16}$ in memory) *With big-endian, the byte order would be* 01 23 45 67*, while with little-endian the order would be* 67 45 23 01*. If x is stored starting at address p, then address p would contain the byte* 01 *or* 67 *if x is stored in respectively big-endian and little-endian order.*

Note that byte ordering does not affect the ordering of digits (or bits) *within* each byte. This is because individual bits do not have addresses; only bytes do.

While big-endian used to be dominant, most current machines use little-endian byte order. This can be confusing when inspecting raw memory contents, as the convention in mathematical notation is big-endian. Fortunately, byte order is unimportant in most programming tasks: it is handled by the programming language implementation. We can only observe the byte order when we explicitly decompose values into their constituent bytes. Bit-shifting integers and similar always works as expected.

4.2.2 Addresses and pointers

The addresses used to index memory are unsigned numbers of a fixed size that depends on the computer architecture. 32-bit addresses used to be common, but most mainstream computers now use 64-bit addresses. The size of the address is independent of how much memory is actually physically installed in the computer—with 64 bits we can address a memory comprising 2^{64} bytes, which is *far* larger than any current or planned computer.[2]

[2] As of this writing, the largest supercomputer in the world is the Japanese Fukagu, which contains approximately 2^{52} bytes of memory in total, although this is spread across 158976 distinct physical memories.

As a model, we can imagine that addresses beyond the physical capacity of the computer result in an error. For example, in a computer with $2^{30}B = 1GiB$ of memory, addresses of 2^{30} and above would be invalid. The consequences of accessing an invalid address varies depending on the machines, but in most cases the program will be terminated—under Unix-like operating systems, this is called a *segmentation fault*. You will see lots of these in your own C programs.

To the computer, an address is merely an unsigned number, and programming directly with them is notoriously error-prone. Most languages do not expose addresses at all, and even those that do tend to give them a distinct type to avoid mistakes. In C, we can use the `&` prefix operator to take the address of a variable `x`, as follows:

```
&x
```

If `x` has type `T`, then `&x` has type `T*`, read as "pointer to `T`". In C, a variable that contains an address is called a *pointer*. We can *dereference* a pointer by using the prefix operator `*`. This means taking the value of the pointer (which is an address) and retrieving the value at that address. For example, if `px` has type `T*`, the expression `*px` has type `T`. Be careful not to confuse the use of `*` in *types* (where it *adds* a layer of indirection) with the use of `*` in *expressions* (where it *removes* a layer of indirection). Pointers are an inherently difficult topic, but flaws in C's syntax makes it no easier.

If you print the addresses used for variables in your own programs, you will see that many have values that far exceed the amount of physical memory in your computer. This is due to virtual memory, which we will return to later, but means there is a decoupling between the addresses seen by software and the actual physical addresses used by hardware. As a model, we can see program memory not as an array with contiguous valid addresses, but instead as an array with "gaps" of invalid addresses.

4.3 Text and Characters

Computers understand only bits, aggregated into bytes. These are stored as voltage differences inside electronic circuitry. In order for a computer to be useful, this data must be made comprehensible to a human.

This usually happens through an *input-output* (IO) device. Imagine an electronic typewriter that receives bytes over a wire. Upon receiving a byte, it consults a table called a *coded character set* that maps each of the 256 possible bytes to a character, and then prints the corresponding character on a piece of paper. This was exactly how early *teletypes* worked, and the idea of associating numbers with characters, which are then printed or shown as appropriate, is still relevant. Today, it is unlikely that you interact with your computer through a teletype, and instead the bytes are passed through multiple layers of hardware and software until your monitor ultimately illuminates multiple tiny LEDs to form the desired shape on the screen.

Control characters				Normal characters											
000	nul	016	dle	032	␣	048	0	064	@	080	P	096	`	112	p
001	soh	017	dc1	033	!	049	1	065	A	081	Q	097	a	113	q
002	stx	018	dc2	034	``	050	2	066	B	082	R	098	b	114	r
003	etx	019	dc3	035	#	051	3	067	C	083	S	099	c	115	s
004	eot	020	dc4	036	$	052	4	068	D	084	T	100	d	116	t
005	enq	021	nak	037	%	053	5	069	E	085	U	101	e	117	u
006	ack	022	syn	038	&	054	6	070	F	086	V	102	f	118	v
007	bel	023	etb	039	'	055	7	071	G	087	W	103	g	119	w
008	bs	024	can	040	(	056	8	072	H	088	X	104	h	120	x
009	tab	025	em	041	)	057	9	073	I	089	Y	105	i	121	y
010	lf	026	eof	042	*	058	:	074	J	090	Z	106	j	122	z
011	vt	027	esc	043	+	059	;	075	K	091	[	107	k	123	{
012	np	028	fs	044	'	060	<	076	L	092	␣	108	l	124	|
013	cr	029	gs	045	−	061	=	077	M	093	]	109	m	125	}
014	so	030	rs	046	.	062	>	078	N	094	ˆ	110	n	126	˜
015	si	031	us	047	/	063	?	079	O	095	_	111	o	127	del

Table 4.1: The ASCII table, mapping decimal numbers to the corresponding character. Note that not all of these characters are intended for humans—some are control characters for operating the teletype or otherwise controlling the communication link. While some of these are still used, mainly the ones corresponding to whitespace, most are rarely seen today.

Definition 4.3 (Coded character set) *A mapping from integers to characters.*

Many character sets used to proliferate, but *ASCII* eventually became dominant. ASCII, shown on table 4.1 defines only 127 characters, leaving the 8th bit free. Originally this 8th bit was used to perform error correction when data was transmitted across noisy telephone lines, but later it was used to extend the ASCII-table with language-specific variants. For example, the ISO-8859-1 character set added support for most Western European characters. Countries that did not use the Latin alphabet (such as Russia, China, or Japan) defined their own non-ASCII character sets. As always, data was not self-describing, so in order to interpret a byte sequence as text, you had to know which character set was used to encode it.

Eventually, the Unicode standard was established, which was conceived as a single character set that could encompass *all* the worlds languages. In ASCII and its variants, each byte corresponds to a single character, but this is insufficient for Unicode, which as of this writing defines 149,186 characters[3]. While Unicode is still rooted in the idea of mapping numbers to characters, it provides multiple ways of encoding each number as (usually multiple) bytes. The most common encoding, called UTF-8, is a variable-width encoding where the number of bytes per character depends on the character being encoded. However, for the subset of Unicode corresponding to ASCII, UTF-8 has the property that any ASCII text is also valid UTF-8, and encodes the same characters. For simplicity, we will stick to ASCII in the following.

[3]For a very wide definition of "character"—everything from hieroglyphs, right-to-left indications, combining accents, and emoji are part of Unicode.

4.3.1 Numbers and Text

When we want to print a number, we have to produce a byte sequence corresponding to an encoding of the digits. For example, the number

$$1234_{10} = 04d2_{16}$$

might be stored like this in memory, using decimal notation, 2 bytes, and little-endian representation:

210	4

We cannot merely submit these two bytes to the teletype. The number 210 does not correspond to any ASCII character at all, and 4 corresponds to the control character `eot` (for "end of transmission"). Instead, we must convert it to the byte string

49	50	51	52

which we can then submit to the teletype or write to a text file. Note that the number 49 encodes the ASCII digit `1`, 50 encodes `2`, etc. It is an easy mistake to conflate the number in memory with the (in principle arbitrary) number that identifies a *digit character* in the ASCII table. Note also that endianness does not matter for human-readable text. The order in which the characters appear in memory (or the order in which they are submitted to the IO device) are also the order in which they will appear to a human.[4]

4.3.2 Strings in C

A sequence of characters is colloquially called a *string* in most programming languages, and is typically one of the most central types. Unfortunately, C has no built-in string type. Instead C represents strings with the type `char*`, representing the address of the first character in the string, and indicates the end of the string with a single byte with the value 0.

This representation, called zero-termination, is recognised as one of the greatest design mistakes in computer science and has been the cause of countless security holes. The reason is that it is awfully easy to forget to add the trailing 0 byte when constructing a string. When later code then attempts to work on the string, it will go past the intended end and into arbitrary memory, until a 0 byte happens to be encountered.

Most other languages employ a representation where a string is stored alongside an integer containing the size of the string. This is much less error-prone.

[4]There are character encodings that represent each characters using a fixed number of bytes, such as UTF-32 which uses four bytes per character. These are subject to byte ordering issues *within* each character. Text using this encoding is usually preceded by a *byte order mark* indicating the endianess used. This is not a problem for UTF-8. If you make correct choices in life, you will never have to worry about this.

Chapter 5

Compiled and Interpreted Languages

A computer can directly execute only machine code, consisting of raw numeric data. Machine code can be written by humans, but we usually use symbolic *assembly languages* to make it more approachable. However, even when using an assembly language, this form of programming is very tedious. This is because assembly languages are (almost) a transparent layer of syntax on top of the raw machine code, and the machine code has been designed to be efficient to *execute*, not to be a pleasant programming experience. Specifically, we are programming at a very low level of abstraction when we use assembly languages, and with no good ability to build new abstractions. In practice, almost all progamming is conducted in *high-level languages*.

5.1 Low-level and High-Level Languages

For the purpose of this chapter, a high-level programming language is a language that is designed not to directly represent the capabilities and details of some machine, but rather to *abstract* the mechanical details, in order to make programming simpler. However, we should note that "high-level" is a spectrum. In general, the meaning of the term "high-level programming language" depends on the speaker and the context (fig. 5.1). The pioneering computer scientist Alan Perlis said: *"A programming language is low-level when its programs require attention to the irrelevant"*. During the course you will gain familiarity with the programming language C, which *definitely* requires you to pay attention to things that are often considered irrelevant, which makes it low-level in Perlis's eyes. However, we will see that the control offered by C provides some capabilities, mostly the ability to *tune* our code for high performance, that are for some problems *not* irrelevant. The term *mid-level programming language* might be a good description of C, as it fills a niche between low-level assembly languages, and high-level languages such as Python and F#.

Generally speaking, low-level languages tend to be more *difficult* to program

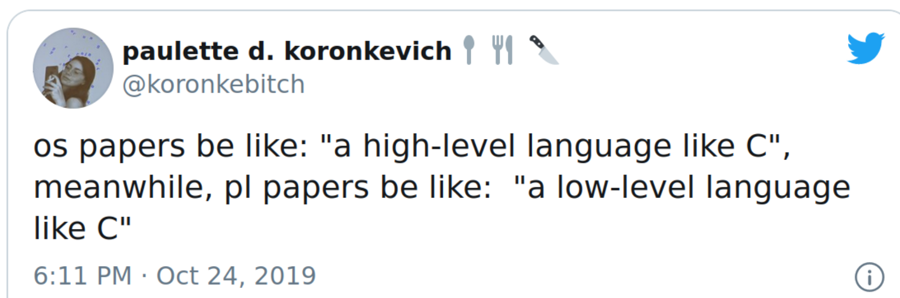


Figure 5.1: A remark on the clarity of terms in computer science.

in, while offering greater potential *performance* (i.e. programs written in them are faster). Higher-level languages are much easier to program in, but run slower and require more machine resources (e.g. memory). Given the speed of modern computers, this is a price we are often willing to pay—especially in the common case where the slowest part of our program is waiting for information from disk or network. Do not make the mistake of assuming that a program written in a low-level language is *always* faster than one written in a high-level language. Choice of algorithm is often more important than choice of language. Further, some high-level languages are specifically designed to execute very efficiently. But there is no free lunch: these languages make tradeoffs in other areas. There is no objective measure of where a language lies on the scale of "level-ness", so while a statement such as *"Python is more high-level than C"* is unlikely to raise any objections, it is usually pointless to try to rank very similar languages on this spectrum.

5.2 Compilers and Interpreters

As the computer natively understands only its machine code, other languages must be *translated* to machine code in order to run. For historical reasons, this is called *compilation*. We say that a compiler takes as input a file with a *source program*, and produces a file containing an executable *machine program* that can be run directly. This is a very simplified model, for the following reasons:

1. Strictly speaking, a compiler does not have to produce machine code. A compiler can also produce code in a different high level languag. For example, with the rise of browsers, it has become common to write compilers that produce JavaScript code.

2. The machine program normally cannot be *directly* executed, as modern systems have many layers of abstraction on top of the processor. While the compiler does produce machine code, it is usually stored in a special file format that is understood by the *operating system*, which is responsible for making the machine code available to the processor.

3. The actual compiler contains many internal steps. Further, large programs are typically not compiled all at once, but rather in chunks. Typically, each *source file* is compiled to one *object file*, which are finally *linked* to form an executable program.

While compilers are a fascinating subject in their own right, we will discuss them only at a superficial level. For a more in-depth treatment, you are encouraged to read a book such as Torben Mogensen's *Basics of Compiler Design*[1].

In contrast, an *interpreter* is a program that executes code directly, without first translating it. The interpreter can be a compiled program, or itself be interpreted. At the bottom level, we always have a CPU executing machine code, but there is no fundamental limit to how many layers of interpreters we can build on top. However, the most common case is that the interpreter is a machine code program, typically produced by a compiler. For example, Python is an interpreted language, and the `python` interpreter program used by most people is written in C, and compiled to machine code.

Interpreters are generally easier to construct than compilers, especially for very dynamic languages such as Python. The downside is that code that is interpreted generally runs much slower than machine code. This is called the *interpretive overhead*. When a C compiler encounters an integer expression `x + y`, then this can likely be translated to a single machine code instruction—possibly preceded by instructions to read `x` and `y` from memory. In contrast, whenever the Python interpreter encounters this expression, it has to analyse it and figure out what is supposed to happen (integer addition), and then dispatch to an implementation of that operation. This is usually at least an order of magnitude slower than actually doing the work. This means that interpreted languages are usually slower than compiled languages. However, many programs spend most of their time waiting for user input, for a network request, or for data from the file system. Such programs are not greatly affected by interpretive overhead.

As an example of interpretive overhead, let us try writing programs for investigating the Collatz conjecture. The Collatz conjecture states that if we repeatedly apply the function

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

to some initial number greater than 1, then we will eventually reach 1. To investigate this function, the Python program `collatz.py` in listing 5.1 takes an initial k from the user, then for every $1 \leq n < k$ prints out n followed by the number of iterations of the function it takes to reach 1.

[1] `http://hjemmesider.diku.dk/~torbenm/Basics/`

Listing 5.1: A Python program for investigating the Collatz conjecture.

```
import sys

def collatz(n):
    i = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        i = i + 1
    return i

k = int(sys.argv[1])
for n in range(1, k):
    print(n, collatz(n))
```

In a Unix shell we can time the program for $k = 100000$ as follows, where we explicitly ignore the output[2]:

```
$ time python3 ./collatz.py 100000 >/dev/null

real    0m1.368s
user    0m1.361s
sys     0m0.007s
```

The `real` measurement tells us that the program took a little more than $1.3s$ to run in the real world (we'll talk about the difference between `user` and `sys` in chapter 10).

Now let us consider the same program, but written in C, which we call `collatz.c`, and is shown in listing 5.2.

C is a compiled language, so we have to compile `collatz.c`:

```
$ gcc collatz.c -o collatz
```

And then we can run it:

```
$ time ./collatz 100000 >/dev/null

real    0m0.032s
user    0m0.030s
sys     0m0.002s
```

[2]This is a very naive way of timing programs—it's adequate for programs that run for a relatively long time, but later we will have to discuss better ways to measure performance. In particular, it includes the overhead of starting up the Python interpreter, and it is sensitive to noise, because we only take a single measurement.

Listing 5.2: A C program for investigating the Collatz conjecture.

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

int collatz(int n) {
  int i = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    i++;
  }
  return i;
}

int main(int argc, char** argv) {
  int k = atoi(argv[1]);
  for (int n = 1; n < k; n++) {
    printf("%d␣%d\n", n, collatz(n));
  }
}
```

Only $0.032s$! This means that our C program is

$$\frac{1.368}{0.032} = 42.75$$

times faster than the Python program[3]. This is not unexpected. The ease of use of interpreted languages comes at a significant overhead.

5.2.1 Advantages of interpreters

People implement interpreters because they are easy to construct, especially for advanced or dynamic languages, and because they are easier to work with. For example, when we are compiling a program to machine code, the compiler discards information about the source code, which makes it difficult to relate the generated machine code with the code originally written by the human. This makes debugging harder, because the connection between what the machine physically *does*, and what the programmer *wrote*, is not explicit. In contrast, an interpreter more or less executes the program as written by the program-

[3]This is the *speedup in latency*—a concept we will return to in section 11.1.1

mer, so when things go wrong, it is easier to explain where in the source code the problem occurs.

In practice, to help with debugging, good compilers can generate significant amounts of extra information in order to let special *debugger* programs map the generated machine code to the original source code. However, this does tend to affect the performance of the generated code.

Another typical advantage of interpreters is that they are straightforwardly *portable*. When writing a compiler that generates machine code, we must explicitly write a code generator every CPU architecture we wish to target. An interpreter can be written once in a portable programming language (say, C), and then compiled to any architecture for which we have a C compiler (which is essentially all of them).

As a rule of thumb, very high-level languages tend to be interpreted, and low-level languages are almost always compiled. Unfortunately, things are not always so clear cut in practice, and *any* language can in principle be compiled—it may just be very difficult for some languages.

5.2.2 Blurring the lines

Very few production languages are *pure* interpreters, in the sense that they do no processing of the source program before executing it. Even Python, which is our main example of an interpreted language, does in fact compile Python source code to Python *bytecode*, which is a kind of invented machine code that is then interpreted by the Python *virtual machine*, which is an interpreter written in C. We can in fact ask Python to show us the bytecode corresponding to a function:

```
>>> import dis
>>> def add(a,b,c):
...     return a + b + c
...
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 LOAD_FAST                2 (c)
              8 BINARY_ADD
             10 RETURN_VALUE
```

This is not machine code for any processor that has ever been physically constructed, but rather an invented machine code that is interpreted by Python's bytecode interpreter. This is a common design because it is faster than interpreting raw Python source code, but it is still much slower than true machine code.

5.2.2.1 JIT Compilation

An even more advanced implementation technique is *just-in-time* (JIT) compilation, which is notably used for languages such as C#, F# and JavaScript. Here, the source program is first compiled to some kind of intermediary bytecode, but this bytecode is then further compiled *at run-time* to actual machine code. The technique is called just-in-time compilation because the final compilation typically occurs on the user's own machine, immediately prior to the program running.

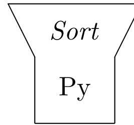
The main advantage of JIT compilation is that programs run much faster than when interpreting bytecode, because we ultimately do end up executing a machine code version of the program. Because JIT compilation takes place while the program is running, it is also able to inspect the actual run-time behaviour of the program and tailor the code generation to the actual data encountered in use. This is useful for highly dynamic languages, where traditional *ahead-of-time* (AOT) compilers have difficulty generating good code. In theory, a JIT compiler can always be *at least as good* as an AOT compiler, but in practice, AOT compilers tend to generate better code, as they can afford to spend more time on compilation. In practice, JIT compilers are only used to compute those parts of the program that are "hot" (where a lot of time is spent), and an interpreter is used for the rest. This tends to works well in practice, due to the maxim that 80% of the run-time is spent in 20% of the code. An AOT compiler will not know which 20% of the code is actually hot, and so must dedicate equal effort to every part, while a JIT compiler can measure the run-time behaviour of the program, and see where it is worth putting in extra effort.

The main downside of JIT compilation is that it is difficult to implement. It has been claimed that AOT compilers are 10× as difficult to write as interpreters, and JIT compilers are 10× as difficult to write as AOT compilers.

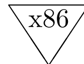
5.3 Tombstone Diagrams

Interpreters and compilers allow us to consider programs as input and output of other programs. That is, they are *data*. *Tombstone diagrams* (sometimes called *T-diagrams*) are a visual notation that lets us describe how a program is translated between different languages (*compiled*), and when execution takes place (either through a software interpreter or a hardware processor). They are not a completely formal notation, nor can they express every kind of translation or execution, but they are useful for gaining an appreciation of the big picture.

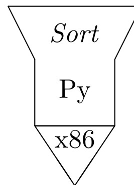
As the most fundamental concept, we have programs, which are written in some language. Suppose we have a sorting program written in Python, which we draw as follows:



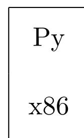
This is an incomplete diagram, since it contains programs we have not described how to execute. A machine that executes some language, say x86 machine code is illustrated as a downward-pointing triangle:



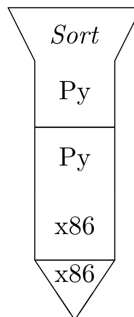
We can say that the Python program is executed on this machine, by stacking them:



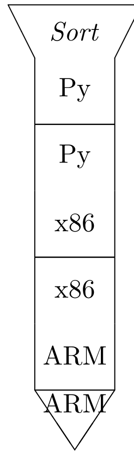
But this diagram is *wrong* — we are saying that a program written in Python is running on a machine that executes only x86. When putting together a tombstone diagram, we must ensure that the languages of the components match. While on paper we can just assume a Python machine, this is not very realistic. Instead, we use an interpreter for Python, written in x86, which as a tombstone diagram is drawn like this:



We can then stack the Python program on top of the interpreter, which we then stack on top of the x86 machine:

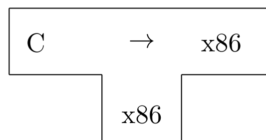


But maybe we are actually running on an ARM machine (as can be found in most phones), but still only have a Python interpreter in x86. As long as we have an x86 interpreter written in ARM machine code, this is no problem:

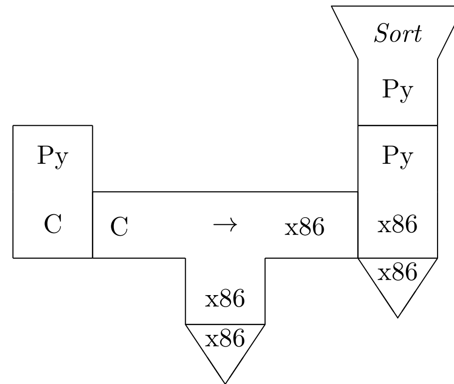


There is no limit to how tall we can stack interpreters. All that matters is that at the end, we have either a machine that can run the implementation language of the bottommost interpreter. Of course, in practice, each level of interpretation adds overhead, so while tombstone diagrams show what is *possible*, they do not necessarily show what is a good idea. Tall interpreter stacks mostly occur in retrocomputing or data archaeology, where we are simulating otherwise dead hardware.

The diagrams above are a bit misleading, because the Python interpreter is not actually written in machine code—it is written in C, which is then translated by a compiler. With a tombstone diagram, a compiler from C to x86, where the compiler is itself also written in x86, is illustrated as follows:

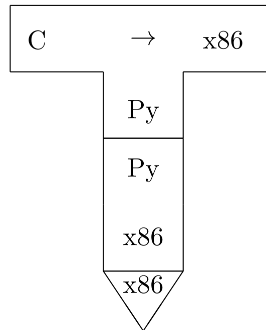


We can now put together a full diagram showing how the Python interpreter is translated from C to x86, and then used to run a Python program:



For a diagram to be valid, every program, interpreter, or compiler, must either be stacked on top of an interpreter or machine, or must be to the left of a compiler, as with the Python interpreter above.

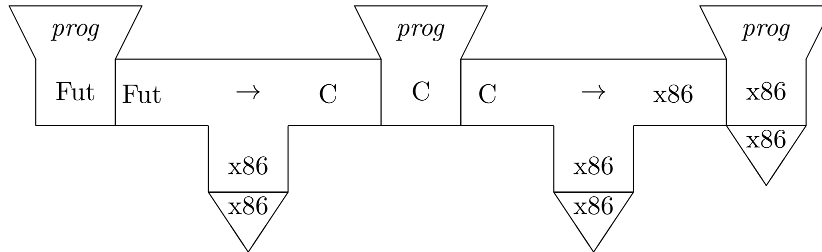
Compilers are also just programs, and must either be executed directly by an appropriate machine, or interpreted. For example, the following diagram shows how to run a C compiler in Python, on top of a Python interpreter in x86 machine code:



How the Python interpreter has been obtained, whether written by hand or compiled from another language, is not visible in the diagram.

We can also use diagrams to show compilation pipelines that chain multiple compilers. For example, programs written in the Futhark[4] programming language are typically compiled first to C, and then uses a C compiler to generate machine code, which we can then finally run:

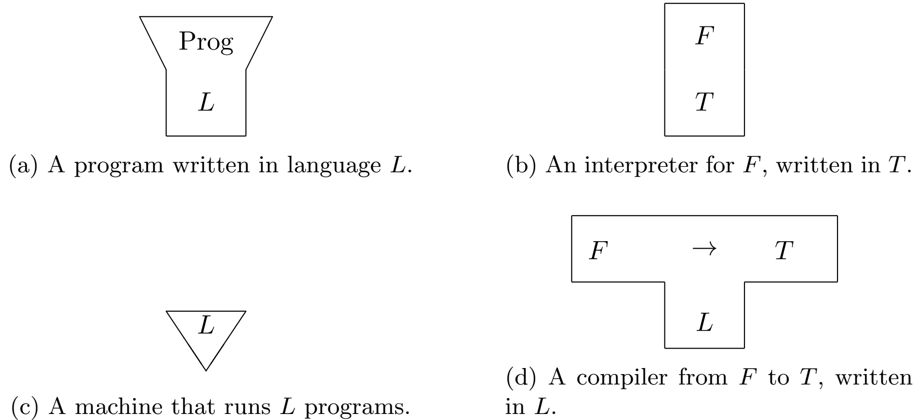
[4] `https://futhark-lang.org`



Many compilers have multiple internal steps—for example, a C compiler does not usually generate machine code directly, but rather generates symbolic assembly code, which an *assembler* then translates to binary machine code. Typically tombstone diagrams do not include such details, but we can include them if we wish, such as with the Futhark compiler above.

Tombstone diagrams can get awkward in complex cases (sometimes there will be no room!), but they can be a useful illustration of complex setups of compilers and interpreters. Also, if we loosen the definition of "machine" to include "operating systems", then we can use these diagrams to show how we can emulate Windows or DOS programs on a GNU/Linux system.

Tombstone diagrams hide many details that we normally consider important. For example, a JIT compiler is simply considered an interpreter in a tombstone diagram, since that is how it appears to the outside. Also, tombstone diagrams cannot easily express programs written in multiple languages, like the example shown in section 5.4. Always be aware that tombstone diagrams are a very high-level visualisation. In practice, such diagrams are mostly used for describing *bootstrapping* processes, by which we make compilers available on new machines. The tombstone diagram components are summarised in fig. 5.2.



(a) A program written in language L.

(b) An interpreter for F, written in T.

(c) A machine that runs L programs.

(d) A compiler from F to T, written in L.

Figure 5.2: A summary of tombstone diagram building blocks.

5.4 Combining Python and C

As discussed above, interpreted languages are typically substantially slower than compiled languages, especially for languages with high *computational intensity*. By this term, we mean how much of the execution time is spent directly executing program code, and how much is spent waiting for data (e.g. user input or network data). For programs with low computational intensity, an interpreted language like Python is an excellent choice, as the interpretive overhead has little impact. However, Python is also very widely used for computationally heavy programs, such as data analysis. Do we just accept that these programs are much slower than a corresponding program written in C? Not exactly. Instead, we use high-performance languages, such as C, to write *computational kernels* in the form of C functions. These C functions can then be called from Python using a so-called *foreign function interface* (FFI).

As a simple example, let us consider the `collatz.c` program from listing 5.2. Instead of compiling the C program to an executable, we compile it to a so-called *shared object*, which allows it to be loaded by Python[5]:

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

We can now write a Python program that uses the `ctypes` library to access the compiled code in the `libcollatz.so` file, and call the `collatz` function we wrote in C:

Listing 5.3: A Python program that uses a C implementation of `collatz`.

```
import ctypes
import sys

c_lib = ctypes.CDLL('./libcollatz.so')

k = int(sys.argv[1])
for n in range(1, k):
    print(n, c_lib.collatz(n))
```

Let's time it as before:

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null

real    0m0.165s
user    0m0.163s
sys     0m0.003s
```

The pure Python program ran in $1.3s$, the pure C in $0.032s$, and this mixture in $0.165s$ - significantly faster than Python, but slower than C by itself. The difference is mostly down to the work required to convert Python values to C

[5]Don't worry about the details of the command line options here—the technical details are less important than the concept.

values when calling `c_lib.collatz`. The overhead is particularly acute for this program, because each call to `collatz` does relatively little work.

While this example is very simple, the basic idea is *fundamental* to Python's current status as perhaps the most popular language for working data scientists and students. Ubiquitous libraries such as NumPy and SciPy have their computational core written in high-performance C or Fortran, which is then exposed in a user-friendly way through Python functions and objects. While a program that uses NumPy is certainly much slower than a tightly optimised C program, it is *much* faster than a pure Python program would be, and *far* easier to write than a corresponding C program.

Chapter 6

Data Layout

One of the things that makes C a difficult programming language is that it does not provide many built-in data types, and provides poor support for making it convenient to work with new data types. In particular, C has notoriously poor support for multi-dimensional arrays. Given that multi-dimensional arrays are perhaps the single most important data structure for scientific computing, this is not good. In this chapter we will look at how we *encode* mathematical objects such as matrices (two-dimensional arrays) with the tools that C makes available to us. One key point is that there are often multiple ways to represent the same object, with different pros and cons, depending on the situation.

6.1 Arrays in C

At the surface level, C does support arrays. We can declare a $n \times m$ array as

```
double A[n][m];
```

and then use the fairly straightforward `A[i][j]` syntax to read a given element. However, C's arrays are a second-class language construct in many ways:

- They decay to pointers in many situations.
- They cannot be passed to a function without "losing" their size.
- They cannot be returned from a function at all.

In practice, we tend to only use language-level arrays in very simple cases, where the sizes are statically known, and they are not passed to or from functions. For general-purpose usage, we instead build our own representation of multi-dimensional arrays, using C's support for *pointers* and *dynamic allocation*. Since actual machine memory is essentially a single-dimensional array, working with multi-dimensional arrays in C really just requires us to answer one central question:

How do we map a multi-dimensional index to a single-dimensional index?

Or to put it another way, representing a d-dimensional array in C requires us to define a bijective[1] *index function*

$$I : \mathbb{N}^d \to \mathbb{N} \tag{6.1}$$

The index function maps from our (mathematical, conceptual) multi-dimensional space to the one-dimensional memory space offered by an actual computer. This is sometimes also called *unranking*, although this is strictly speaking a more general term from combinatorics.

As an example, suppose we wish to represent the following 3×4 matrix in memory:

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix} \tag{6.2}$$

We can do this in any baroque way we wish, but the two most common representations are:

Row-major order, where elements of each *row* are contiguous in memory:

11	12	13	14	21	22	23	24	31	32	33	34

with index function

$$(i, j) \mapsto i \times 4 + j$$

Column-major order, where elements of each *column* are contiguous in memory:

11	21	31	12	22	32	13	23	33	14	24	34

with index function

$$(i, j) \mapsto j \times 3 + i$$

The index functions are generalised on fig. 6.1. Note that the two representations contain the exact same values, so they encode the same mathematical object, but in different ways. The intuition for the row-major index function is that we first skip i rows ahead to get to the row of interest, then move j columns into the row.

Row-major order is used by default in most programming languages and libraries, but not universally so—the scientific language Fortran is famously column-major. The NumPy library for Python uses row-major by default (called `C` in Numpy), but one can explicitly ask for arrays in column-major order (called `F`), which is sometimes needed when exchanging data with systems that expect a different representation.

[1] A bijective function is a function between two sets that maps each element of each set to a distinct element of the other set.

$$(i, j) \mapsto i \times m + j \quad (6.3)$$

(a) Row-major indexing.

$$(i, j) \mapsto j \times n + i \quad (6.4)$$

(b) Column-major indexing.

Figure 6.1: Index functions for $n \times m$ arrays represented in row-major and column-major order. For an example of why computer scientists tend to prefer 0-indexing, try rewriting the above to work with 1-index arrays instead.

6.1.1 Implementation in C

Let's look at how to implement this in C. Let's say we wish to represent the matrix from eq. (6.2) in row-major order. Then we would write the following (assuming `n=3`, `m=4`):

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 12;
...
A[11] = 34;
```

Note that even though we *conceptually* wish to represent a two-dimensional array, the actual C type is technically a single-dimensional array with 12 elements. If we when wish to index at position (i, j) we then use the expression `A[i*4+j]`.

Similarly, if we wished to use column-major order, we would program as follows:

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 21;
...
A[11] = 34;
```

To C there is no difference—and there is no indication in the types what we intended. This makes it very easy to make mistakes.

Note also how it is on *us* to keep track of the sizes of the array—C is no help. Don't make the mistake of thinking that `sizeof(A)` will tell you how big this array is—while C will produce a number for you, it willindicate the size of a *pointer* (probably 8 on your machine).

Some C programmers like defining functions to help them generate the flat indexes when indexing arrays:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  return i * m + j;
}

int idx2_colmajor(int n, int m, int i, int j) {
  return j * n + i;
}
```

Note how row-major indexing does not use the n parameter, and column-major indexing does not use m.

However, these functions do not on their own fully prevent us from making mistakes. Consider indexing the A array from before with the expression

```
A[idx2_rowmajor(n, m, 2, 5)].
```

Here we are trying to access index $(2,5)$ in a 3×4 array—which is conceptually an out-of-bounds access. However, by the index function, this translates to the flat index $2 \times 3 + 5 = 11$, which is in-bounds for the 12-element array we use for our representation in C. This means that handy tools like valgrind will not even be able to detect our mistake—from C's point of view, we're doing nothing wrong! Things like this make scientific computing in C a risky endeavour. We can protect ourselves by using helper functions like those above, and augment them with assert statements that check for problems:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  assert(i >= 0 && i < n);
  assert(j >= 0 && j < m);
  return i * m + j;
}
```

We can still make mistakes, but at least now they will be noisy, rather than silently reading (or corrupting!) unintended data.

6.1.2 Size passing

With the previously discussed representation, a multidimensional array (e.g. a matrix) is just a pointer to the data, along with metadata about its size. The C language does not help us keep this metadata in sync with reality. When passing one of these arrays to a function, we must manually pass along the sizes, and we must get them right without much help from the compiler. For example, consider a function that sums each row of a (row-major) $n \times m$ array, saving the results to an n-element output array:

Listing 6.1: Summing the rows of a matrix.

```
void sumrows(int n, int m,
             const double *matrix, double *vector) {
  for (int i = 0; i < n; i++) {
    double sum = 0;
    for (int j = 0; j < m; j++) {
      sum += matrix[i*m+j];
    }
    vector[i] = sum;
  }
}
```

C gives us the raw building blocks of efficient computation, but we must put together the pieces ourselves. We protect ourselves by carefully documenting the data layout expected of the various functions. For the `sumrows` function above, we would document that `matrix` is expected to be a row-major array of size $n \times m$.

6.1.3 Slicing

In high-level languages like Python, we can use notation such as `A[i:j]` to extract a *slice* (a contiguous subsequence) of an array, in this case of the elements from index `i` to `j`. No such syntactical niceties are available in C, but by using our knowledge of how arrays are physically laid out in memory, we can obtain a similar effect in many cases.

Suppose `V` is a vector of `n` elements, and we wish to obtain a slice of the elements from index `i` to `j` (the latter exclusive). In Python, we would merely write `V[i:j]`. In C, we compute the size of the slice as

```
int m = j - i;
```

and then compute a pointer to the start of the slice:

```
double *slice = &V[i];
```

Now we can treat `slice` as an `m`-element array that just happens to use the same underlying storage as `V`. This means that we must be careful not to deallocate `V` while `slice` is still in use.

Similarly, if `A` represents a matrix of size `n` by `m` in row-major order, then we can produce a vector representing the `i`th row as follows:

```
double *row = &A[i*m];
```

The restriction is that such slicing can only produce arrays whose elements are *contiguous* in memory. For example, we cannot easily extract a column of a row-major array, because the elements of a column are not contiguous in memory. If we wish to extract a column, then we have to allocate space and copy element-by-element, in a loop[2].

6.1.4 Even higher dimensions

The examples so far have focused on the two-dimensional case. However, the notion of row-major and column-major order generalises just fine to higher dimensions. The key distinction is that in a row-major array, the *last* dimension is contiguous, while for a column-major array, the *first* dimension is contiguous. For a row-major array of shape $n_0 \times \cdots \times n_{d-1}$, the index function where p is a d-dimensional index point is

[2]There are more sophisticated array representations that use *strides* to allow array elements that are not contiguous in memory—NumPy uses these, but they are outside the scope of our course.

$$p \mapsto \sum_{0 \le i < d} p_i \prod_{i < j < d} n_j \tag{6.5}$$

where p_i gets the ith coordinate of p, and the product of an empty series is 1.

Similarly, for a column-major array, the index function is

$$p \mapsto \sum_{0 \le i < d} p_i \prod_{0 \le j < i} n_j \tag{6.6}$$

We can also have more complex cases, such as a three-dimensional array where the two-dimensional "rows" are stored consecutively, but are individually column-major. Such constructions can be useful, but are beyond the scope of this course (and are a nightmare to implement).

Chapter 7

Locality and Caches

For modern computers, computation is a rather cheap operation in terms of time and energy consumption. In fact, it is far more energy-consuming and slow to *move* data than to perform computation. As we discussed in chapter 4, program data is usually stored in memory and must be moved into CPU registers before it can be worked on. Particularly large quantities of data might not even fit in memory, but is stored in local or remote files. The details can differ, but in all cases the situation is the same: data must be moved from *over there* to *here* before it can be used.

This phenomenon that computation is faster than memory, is often called the *memory wall*, and the gap is still widening. While faster processors keep getting designed, memory technology is not keeping up, and therefore we have increasing difficulty supplying our processors with data.

Since data movement is slow and costly compared to computation, a programmer who wants to write high-performance code must be diligent in doing as little of it as possible. This chapter describes the properties of software and hardware that one must exploit to accomplish this.

While we will continue to use C for concrete programming examples, the principles here are truly universal, and inescapable whenever we write code for modern computers. Fortunately, we will see that optimising code to minimise data movement usually doesn't require us to write particularly ugly or complicated code - it is mostly about choosing the right data structures and traversing them in a sensible manner.

7.1 Locality of Reference

Empirically, programs tend to access data located nearby data that was accessed recently. We call this phenomenon the *principle of localty.*

Definition 7.1 (Principle of locality) *Programs tend to access data located near that which was accessed recently.*

Listing 7.1: C code that sums an array.

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

To keep things simple, this chapters is concerned only with data stored in memory, and we will decide if two pieces of data are "close" to each other based on the numerical value of their addresses. For example, the byte at address 100 is considered adjacent to the bytes at addresses 99 and 101. However, some notion of locality is present in any kind of storage technology. Informally, we say that a program "exhibits good locality" when it adheres to the principle of locality.

The principle of locality is what ultimately helps us break through the memory wall. It is a virtuous cycle: we build computers that run programs with good locality quickly, which makes programmers try to improve the locality in their programs.

We distinguish between two kinds of locality.

Definition 7.2 (Temporal locality) *Accessing data that was accessed recently.*

Definition 7.3 (Spatial locality) *Accesing data close to data that was accessed recently.*

While it is possible to run a program, record the exact memory operations performed, and then quantiatively describe how much locality they exhibit, this is often somewhat impractical. Instead, a good programmer should develop the ability to quickly eyeball simple programs and qualitiatively estimate their locality.

7.1.1 Qualitative Estimates of Locality

Consider the program fragment shown in listing 7.1. In terms of data accesses, it accesses the elements of the array `a` serially, with a *stride* of 1 between successive accesses. This is an example of *spatial locality*: we do not access the same data, but we do access data very close to what we accessed in the recent past. Further, we also repeatedly access the variables `sum` and `i`. This is an example of *temporal locality*.

But program data is not the only thing that is stored in memory: the program *code* is as well, in the form of machine instructions. Absent of control flow, instructions are read from memory and executed in the order in which they appear—just like how lines in a C program are executed from the top down. This means they have *spatial locality*. However, we also have a loop in

Listing 7.2: Summing a matrix, iterating along the rows.

```
int sumrows(int A[M][N]) {
  int sum = 0;
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      sum += A[i][j];
  return sum;
}
```

listing 7.1, which means that the same instructions are executed repeatedly. This is *temporal locality*. Generally speaking, program code almost always exhibits good locality, so we tend to ignore it when analysing the locality of a program, and instead focus only on the locality of data accesses.

Further, we focus only on memory accesses. Because simple scalar variables tend to be stored in registers by the C compiler, we look only at how arrays are traversed.

7.1.2 Analysing Array Iteration

Let us consider a more complicated example, seen in listing 7.2. For concision we are use multidimensional C arrays, rather than explicit index functions as in chapter 6. This means that the array `A` is stored in row-major order and that the access `A[i][j]` results in an offset computation `i*N+j`.

Consider the memory accesses performed by the loop. In the first iteration, we have `i=0` and `j=0`, meaning that we access memory at offset `0`. In the next iteration we have `i=0` and `j=1`, meaning we access offset `1`[1]. In every loop iteration, we will access an array element adjacent to the one accessed in the immediately preceding iteration—the stride is 1—and therefore this function exhibits good spatial locality with respect to how it accesses `A`.

Now consider listing 7.3. By repeating the analysis above, we see that the first array element we access is `A[0][0]` and the next one is `A[1][0]`. Since `A` is stored in row-major order, this means we are iterating with a stride of `N`. Assuming `N` is large, this function exhibits *bad* locality with respect to `A`.

7.2 Memory Hierarchies

In an ideal world, memory would be cheap, fast, and plentiful. In the world we actually inhabit, we cannot get all three. In fact, fundamental physical principles mean memory capacity and memory capacity are conflicting properties.

[1]Strictly speaking, we are accessing `1*sizeof(int)`, meaning `4`, but since the first iteration accesses the four bytes stored at offsets `0-3`, this is still considered adjacent. We can ignore the element size for arrays of scalars, but it does have impact if we have arrays containing large compound data structures.

Listing 7.3: Summing the rows of a matrix, iterating along the columns

```
int sumcols(int A[M][N]) {
  int sum = 0;
  for (int j = 0; j < N; j++)
    for (int i = 0; i < M; i++)
      sum += A[i][j];
  return sum;
}
```

Roughly, the larger the capacity of some storage technology, the longer it takes it to retrieve data. However, by exploiting the principle of locality, we can construct a *memory hierarchy* of different storage technologies, starting from the small and fast and ending in the large and slow.

Definition 7.4 (Memory hierarchy) *A separation of computer storage into levels based on their response time and capacity.*

The idea behind a memory hierarchy is that each level contains a *subset* of the data contained in the level below it. Only the level at the bottom contains *all* data. Each level of the hierarchy acts as a *cache* for the one below it.

Definition 7.5 (Cache) *A smaller and faster memory that stores a subset of the contents of a larger and slower memory.*

An example of a memory hierarchy for a computer is shown in fig. 7.1. The exact levels of the memory hierarchy depends on the machine. Further, when using the memory hierarchy model to describe the performance of a system, we often exclude levels that are irrelevant—for example, registers are often ignored because they are controlled by the compiler, and we might ignore anything below main memory if we know that our data will fit.

In this chapter we will focus entirely how the contents of RAM is cached. Further, while real computers usually have multiple levels of caching, we will for simplicity asumme only a single level above main memory, and just call it *the cache*. So in total, we pretend we have only two levels: the small and fast cache on top, and the large and slow main memory below it. In a full memory hierarchy, this situation is simply replicated for each levelall the way down.

7.2.1 Cache Operation

The reason why caching works is due to the principle of locality: most accesses will tend to be towards the top of the hierarchy. When we try to access an address that is already present in the cache, the data is immediately returned to the program.

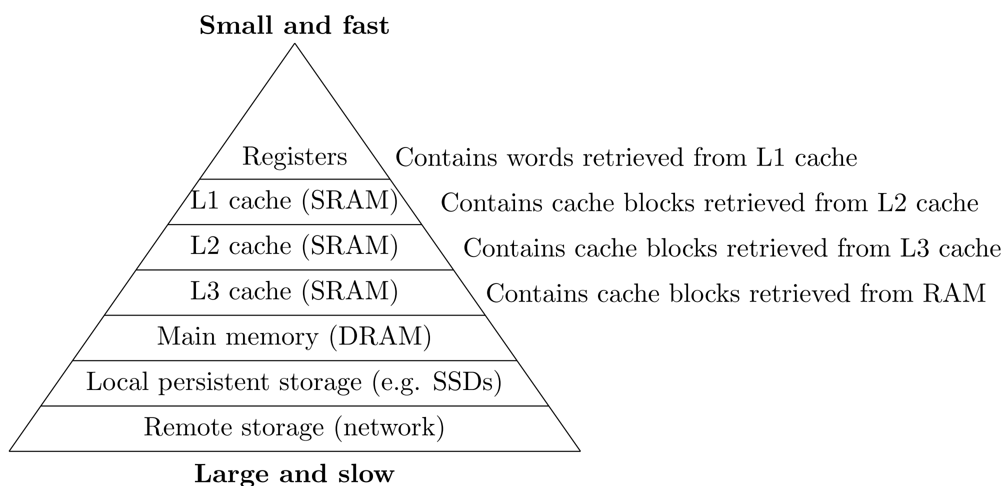


Figure 7.1: Example memory hierarchy for a modern computer. The number of levels and the specifics of their operation depends on the computer in question. Very tiny computers may have no caches at all, but only registers and main memory. Some extremely small processors may have only registers—but these are unlikely to be useful for general-purpose computation.

Definition 7.6 (Cache hit) *A memory operation to an address that is present in the cache.*

If we try to retrieve a piece of data that is not already in the cache, the data is fetched from the next level of the hierarchy, stored in the cache, and then returned to the program.

Definition 7.7 (Cache miss) *A memory operation to an address that is not present in the cache.*

In programs with good locality, most memory accesses will result in cache hits. For such programs, the memory hierarchy provides the illusion of storage that is as large as the bottommost level of the hierarchy, and as fast as the topmost one.

If caches merely contained *exactly* the bytes that had been retrieved previously, then only temporal locality would benefit. Instead, we partition the addressable memory space into *cache blocks*, each containing B bytes.

Definition 7.8 (Cache block) *A B-byte chunk of memory, where B is a power of two.*

Whenever we have a cache miss, the entire block containing the requested address is copied into the cache. This means that subsequent accesses to ad-

dresses that fall within the same block will result in cache hits. This means that programs that exhibit good spatial locality will have more cache hits.

On modern computers, $B = 64$ is common. It is important to be aware that the organisation of memory into cache blocks does not affect addressing: data is still byte-addressed, and programs are not directly aware of B. We merely say that the first B bytes of memory belong to the first cache block, the next B bytes to the second, and so on. This means a memory address x belongs to block $xmodB$. When B is a power of 2 $B = 2^m$, this can be computed simply by dropping the b least significant bits of m. We will return to this in section 7.3. Each level of the memory hierarchy may use different cache block sizes, although they are usually the same for the L1/L2/L3 caches.

Caching is effective when the data being actively actively by the program within some bounded period of time fits within the cache. We call the size of this actively used data the *working set*.

Definition 7.9 (Working set) *The amount of memory accessed by a program within a bounded period of time.*

Definition 7.10 (Memory footprint) *The total amount of memory allocated by a program.*

During the total runtime of a process, different working sets may be in use, as the program switches between different subtasks. Consider a data analysis program sequentially processes a collection of files, one at a time, by loading them into memory. We would characterise the working set of such a program as the size of a *single* file (perhaps the largest one), rather than summing the sizes of all files. We also focus only on data that is being frequently accessed. It is fine for the total memory footprint of a program to exceed what can be stored in the cache, as the excess is simply stored further down in the memory hierarchy. It is not unusual for programs to keep large amounts of data in memory at the same time, while actively working on only small parts at a time. Indeed, this is exactly how to make best use of the memory hierarchy.

Definition 7.11 (Compulsory miss) *A miss that occurs because the cache is empty.*

Definition 7.12 (Capacity miss) *A miss that occurs because the program working set exceeds the cache capacity.*

7.3 Cache Organisation

In this section we will look at how caches are organised in terms of logical units. Their physical representation may be different, but this need not concern us. What matters is how we can characterise the structure of caches and how this affects the way we should program in order to get optimal performance. Beyond storing the actual cache block data, caches must also maintain a variety of book-keeping information so they can quickly determine whether a given

address corresponds to a block that is present. We call a cache block alongside this auxiliary information a *cache line*. when describing the size of a cache, we count only the cache blocks, even though the auxiliary information may take up a significant amount of space.

Definition 7.13 (Cache line) *A structure that contains a cache block as well as metadata about the origin of the block.*

Be aware that this nomenclature is not universally used—many programmers use the term *cache line* interchangeably for both what we call a cache block and a cache line. For clarity, we will use more precise nomenclature.

Specifically, a cache line contains three parts:

1. A cache block containing B bytes.
2. A t-bit *tag* indicating which part of memory is stored in the block. The actual t depends on the specific design of the cache.
3. A *valid bit* indicating whether the cache line contents are sensible.

The reason for the valid bit is that as a piece of hardware, a cache line cannot be "empty". There are always bits stored in the tag and block fields, and they might accidentally correspond to a valid address. The valid bit is used to indicate whether the cache line should be disregarded when checking whether the cache contains a block corresponding to a given address.

At a high level, looking up an address x in a cache then involves computing the tag of the address as

$$x \bmod B$$

and then searching for a cache line with a matching tag field. However, for large caches, searching every single cache line would be slow. It is generally the case that the more choices you have, the slower you are at making a choice. Since the speed of cache lookups is extremely critical for computer performance, this is not something we want to make slower than absolutely necessary.

To solve this, we use *set-associative caches*, where the lines are split into S *sets*, with each set containing L cache lines, such that the total number of cache lines is $S \cdot L$. The bits of an address are then used to determine which set the block corresponding to the address may be stored in. The end result is that we have only L lines to search, rather than all lines in the cache.

Definition 7.14 (S-way set associative cache) *A cache with S sets.*

When the number of sets and the block size are both powers of two $S = 2^s, B = 2^b$, we can split a w-bit address into fields, writing x_i for bit i.

$$\underbrace{x_{w-1} \cdots x_{s+b+1}}_{\text{tag}} \underbrace{x_{b+s} \cdots x_s}_{\text{set index}} \underbrace{x_{b-1} \cdots x_0}_{\text{block offset}}$$

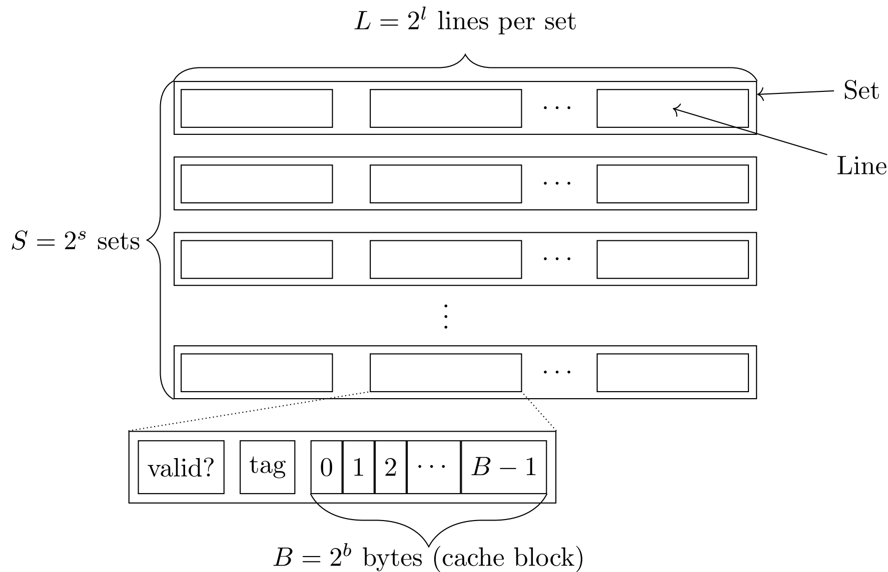


Figure 7.2: Structure of a set-associative cache with S sets, L lines per set, and a B-byte block per line, where we assume all of these are powers of 2. The total size of this cache is $S \times L \times B$.

Example 7.1 (Fields for an 8-bit address scheme) *Consider an* 8*-bit address on a system with a block size of* $B = 2^2$ *and* $S = 2^3$ *sets, with. Going from right to left, the first* $b = 2$ *bits constitute the offset, the next* $s = 3$ *bits the set index, and the remainder are the tag.*

$$\underbrace{x_7x_6x_5}_{tag}\underbrace{x_4x_3x_2}_{tag}\underbrace{x_1x_0}_{offset}$$

In a set-associative cache, a cache block is limited to being stored in only a subset of all cache lines. This means we risk having cache misses even through the working set of the program is smaller than the total size of the cache. This is called a *conflict miss*.

Definition 7.15 (Conflict miss) *A miss that occurs because too many blocks of the program working set are mapped to the same cache set.*

The general structure of a set-associative cache is shown in fig. 7.2. When designing a cache, the number of sets must be carefully balanced: too large, and cache lookups take too long. Too small, and we end up with more conflict misses. Balancing this tradeoff is a delicate affair. As of this writing, most CPU caches are 8- or 16-way set associative. Two special cases have dedicated nomenclature.

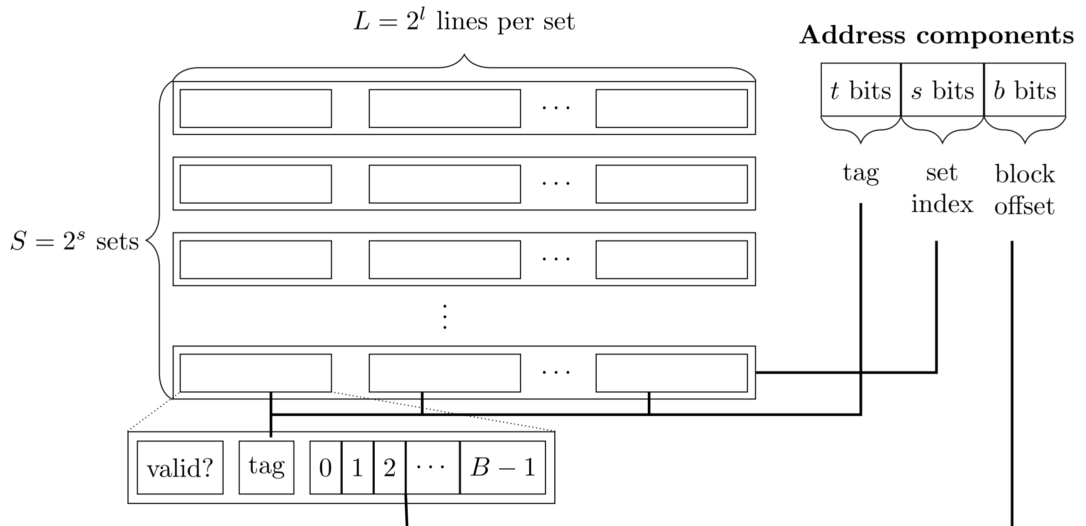


Figure 7.3: Looking up an address in a set-associative cache with S sets and L lines per set. The set index determines which set to look at (in this case, the last one). The set is then searched for a cache line with a tag matching the tag bits of the address. If a match is found (in this case, the first line), and the valid bit is set, the block offset of the address is used to determine which bytes to read from the block.

Definition 7.16 (Direct-mapped cache) *A cache with $L = 1$, meaning that there is only one possible location for a block in the cache.*

Definition 7.17 (Fully associative cache) *A cache with $S = 1$, meaning that a block can be located anywhere in the cache.*

Direct-mapped and fully associative caches tend not to occur for CPU caches, but they appear in lower parts of the memory hierarchy—the latter typically when cache misses are ruinously expensive and so it is worth significant overhead to avoid them.

A diagram of how we perform a cache lookup based on an address is shown in fig. 7.3.

7.4 Cache Performance

To characterise the cache behaviour of a program, we focus on the *miss rate*.

Definition 7.18 (Miss rate) *The fraction of memory references issues by a program not found in the cache.*

For typical programs, the miss rate for the L1 cache is often around $3 - 10\%$, and often less than 1% for L2 and below.

Beyond the miss rate, we are also often interested in the *hit time* and *miss penalty*.

Definition 7.19 (Hit time) *Time taken in the event of a cache hit.*

Definition 7.20 (Miss penalty) *Time taken in the event of a cache miss.*

A cache hit is typically very fast—perhaps 4 clock cycles for L1, 10 for L2, and $45 - 75$ for L3. Conversely, cache misses can be extremely costly—easily hundreds of cycles when we need to go all the way to main memory. This means that a cache miss can be two orders of magnitude slower than a cache hit, and that minimising cache misses is therefore crucial for performance.

7.4.1 Writing cache-friendly code

When writing cache-friendly code, and for that matter whenever we want to optimise the performance of a program, we focus on the innermost loops. In a typical program, the majority of the run-time is spent in a comparatively small portion of the overall code. Generally, it is rarely worth optimising non-loop code.

As a general rule of thumb for writing cache-friendly code, we should avoid dereferencing many different pointers, as we have in principle no way of knowing where in the memory they point. If we perform a huge number of tiny allocations with `malloc()`, we have no knowledge of where in memory they are located relative to each other, and no way of ensuring that we use them in an order that provides good spatial locality. As an example, linked lists have notoriously terrible cache behaviour, because neighbouring links in the list may be arbitrarily distant in memory. Traversing a linked list can easily cause a cache miss whenever we moved to the next element. It is better to work on a few large allocations (such as arrays), where we can analyse the access patterns and decide whether our accesses exhibit good locality.

Another rule of thumb is to minimise the program footprint. If our working set fits in L3 cache, then we will never see a L3 cache miss after the initial compulsory misses needed to load the data into cache in the first place. Sometimes this can be achieved by using smaller data types (e.g. 16-bit integers instead of 32-bit integers) or by clever representations that take advantage of *sparsity* (e.g. if a large matrix is mostly zeroes, store only the non-zero elements as well as their coordinates).

But sometimes we simply have to work on a large piece of data that will not fit in cache. We must carefully consider how we traverse it to minimise the number of cache misses. A common example is matrix multiplication. As matrix multiplication is a fundamental primitive in many numerical techniques, much time has been spent developing elaborate implementation techniques for making them run as fast as possible on pretty much any computer ever built. Our analysis, and our optimisation, will be comparatively simple, but hopefully still illustrative.

Listing 7.4: Matrix multiplication in C.

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

Our starting point is the program fragment shown in listing 7.4. The two `n`-by`n` matrices `a` and `b` are given to us in row-major order, and we must produce a result matrix `c` also in row-major order. Each matrix element is a `double`, comprising 8 bytes. The total amount of work is $O(n^3)$, and each of the n^2 elements in `c` is the result of n summed values.

Further, assume a cache block size

$$B = 64$$

meaning that a single block can hold 8 `doubles`. We also assume that `n` is very large—specifically, so large that the cache is not large enough to hold a full row or column. This means if we traverse a row fully from start to end, then traversing it again cannot take advantage of any cache contents.

We analyse the performance of the program by looking at the access pattern of the innermost loop, because the statements in there are executed n^3 times, making them asymptotically dominant relative to the other statements in the program.

Using the assumptions above, consider a loop that looks as follows:

```
for (int k=0; k<n; k++)
  sum += a[0][k]
```

As we are traversing a single row of `a`, we can take advantage of spatial locality. Each cache miss will load not just the current element into the cache, but a total of 8 elements (ignoring speculative prefetching), meaning that we will only have a cache miss every 8th iteration. This gives us a miss rate of 0.125.

Now consider:

```
for (int k=0; k<n; k++)
  sum += a[k][0]
```

Here we are jumping through memory with a stride of `n`. As we assume `n` is very large (certainly larger than a cache block), we will have a cache miss on every single iteration. The miss rate is 1.

With this line of thinking, we can consider the innermost loop in listing 7.4. The miss rate for accesses to the array `a` is 0.125, to array `b` is 1, and to array `c` is 0, because we access the exact same element in every iteration—providing perfect temporal locality. The total miss rate is 1.125. Intuitively, we are traversing the array `a` row-wise, which is good, and `b` column-wise, which is bad. As a rule of thumb, whenever we index a multi-dimensional array, we are

indexing with a loop variable of the innermost loop, and that loop variable is *not* used to index the last dimension of the array, we are probably traversing in a way that has poor spatial locality.

Listing 7.5: Matrix multiplication in C with the loops permuted.

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

Now consider the program in listing 7.5. It computes the same thing as lst:matmul-kij.c, but with the loops reordered. Looking at the inner loop, we are accessing `c` and `b` row-by-row, so these each have a miss rate of 0.125, while the miss rate for `a` is 0 because we are accessing the same element repeatedly. The total miss rate is 0.25.

Listing 7.6: Matrix multiplication in C with the loops permuted.

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

Can we permute the loops further? Yes, consider listing 7.6. Here we are traversing both `a` and `c` along columns, giving a miss rate of 1, while the accesses to `b` have a miss rate of 0. The total miss rate is 2. Clearly not good.

Our analysis shows that listing 7.5 is the most cache-efficient implementation. Indeed, benchmarking these on a real machine produces fig. 7.4. Note that all we did was permute the loop ordering. While obtaining true peak performance typically requires more complicated program transformations than this, we can often get quite far simply by carefully considering the order in which we traverse our data.

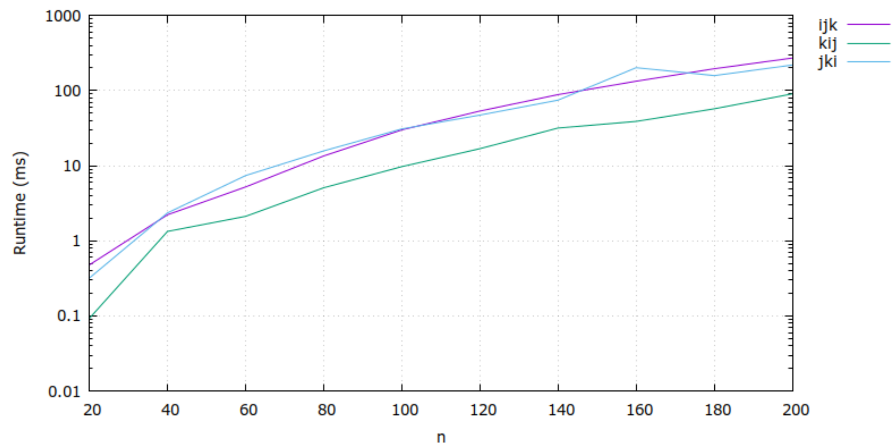


Figure 7.4: Benchmarking the different matrix multiplication implementations. The x axis shows the edge size of the matrices, while the logarithmic y axis shows the runtime in milliseconds. Note that the kij loop ordering, which our analysis determined as optimal, is indeed about a factor of two faster than the other implementations. Somewhat surprisingly, iji (the original program) and jki are about equally fast. This is possibly because the values of `n` used here are not as large as we assumed in our analysis.

Chapter 8

Operating Systems

The operating system (Often abreiviated to OS) is a core piece of software that manages both the device hardware, as well as how other software runs on that hardware. An OS is present on the vast majority of devices we interact with, with the only real exception being simple, dedicated devices that perform only a single function. Most OSs in use today are either explicitly derived from Unix, or are heavily inspired by the systems and operations is developed.

8.1 The Kernel

At its core, the OS provides two abstractions that have proven to be very useful. These are processes and virtual memory. The ultimate aim of both of these concepts is to allow for multiple different programs, users or systems to operate on as many different devices as possible, without programmers having to create custom implementations depending on how their program was run. In order to do this, the OS provides what is known as the kernel. This can be thought of as the very core of the OS, and has complete control over all hardware and software running on a device.

The kernel provides a variety of functions to software running on the same device, which we might think of as being core to how a system operates. These include functions to open files, write to files, read from files, start new processing, end existing processing, and so on. These functions are known as *system calls* are the basic building blocks for any software to interact with any hardware, with a key abstraction being that no software needs to know how to physically write to a file on a given device, it can simply call the *write* function to do so. Crucially, it is only by calling this function that any software can read/write to/from the hardware, giving the kernel complete control over who is writing to what and how they do so.

8.2 Processes

As you should know by now, a CPU can only ever do one thing at a time. A CPU is capable of running a series of instructions, with data being manipulated within registers. Not all data will fit into registers, with the wider data stored in memory. The CPU will progress through its instructions one at a time, starting at the start until it reaches the end. This is not an incorrect model of how a computer works, but is obviously a simplification. One limitation of this system is that the CPU is limited to doing only one thing at a time, yet it should be obvious that your computer is capable of doing many things at the same time. This is enabled by the OS concept of processes.

A process is a conceptual grouping of a programs CPU register counters, the program code, as well as the memory used by the code. This is everything we discussed in our simple example above. This is known as the state, with each processes having its own state of registers, memory and instructions. By taking a complete state and writing it to memory, then writing a new state into the CPU we can swap between two processes. If we do this often enough, the computer can give the impression of doing multiple different things at once, even with only a single CPU. This act of switching states is known as a *context switch* and typically takes at least 1.2 microseconds. This means there is an unavoidable overhead in switching between processes, but its almost always worth it as we can now do multiple things at once.

An example of a process can be almost anything. Our internet browser will run in a process with its own state, whilst a text editor could be another. Games, music players, scripts and apps all run in their own processes with their own state. Often times there are also smaller systems that run in their own background processes, such as update checkers, download managers, network managers and the like. These might not have a dedicated user interface or display but still run in the background, which is why when we inspect our systems using tools such as *top*, *htop* or the *process manager*, we can often see many different processes running on our machine. For most laptops or desktops this process count could even be in the low hundreds. This isn't to say that each process necessarily has something to do at that point, in fact most will be sleeping until they have something to do. For instance, an update checker might only wake once every 24 hours, check with some server if an update has been released, and then go back to sleep.

8.2.1 Concurrency and Parallel

Note that often we aren't literally doing multiple things at once, but that the compute switch between multiple processes so quickly and humans are so slow that it looks like they're taking place at the same time. This is referred to as *concurrency*, where different processes can each progress before any of them have finished, even if they aren't progressing at literally the same time. Modern computers often have more than a single CPU core however, and so they can do multiple things at once. This is referred to as *parallel*, where two things are

progressing at literally the same time. Processes are concurrent, as each is a self contained logical unit, and so we can interrupt any process with another at any point. Anything that is concurrent can be executed in parallel, though this depends on their being the hardware to support it.

8.2.2 Process Scheduling

Any time we introduce some new concept, that presumably is non-trivial for someone to implement, it is always worth asking what it gives us. Processes allow for a complete compartmentalisation of a programs code and its data, from any other programs code and data. As well as allowing for the concurrent and parallel scheduling described above, this compartmentalisation also means that programmers are free to write whatever program they want, as though they had complete control over the entire hardware. Programmers can take as much time as they need, running as many tasks as they wish, without having to worry about who else might need the same resources. Consider your own programming, and what you've asked the computer to do. Presumably at some point you've run a program that has lasted for several hours, potentially performing some complex operations whilst doing so (most modern computer games and even browsers would be another good example of this). You've not need to insert little breaks everyone now and again so that other processes can continue in the background, this is all managed automatically by the OS using processes.

The part of the OS that decides what process to schedule is funnily enough, called a *scheduler*. Process scheduling is actually a vast area of study and expertise in its own right, which we do not have the time to get into here. Instead it suffices to say that most schedulers will try to give each process at least some time on a regular basis to keep executing. This is to avoid what is termed *livelock*, where some execution can progress but the process is not given any resources to do so. Round robin is the most simple scheduling style, where the scheduler will let process 1 run for a short time, then let process 2 run, for a short time, then process 3 and so on, until returning to let process 1 run for a short time again. This keeps everyone moving a little bit, and though there are far more advanced algorithms that start weighting the scheduling based on how much work each process needs to do, this fundamental principle still applies.

8.3 Virtual Memory

The second core concept that OSs provide, is that of virtual memory. Particularly astute readers might have noted that processes can't really enforce their own compartmentalisation, as there is nothing stopping one process from writing into the contents of another. Luckily processes are paired with virtual memory so that everything we've described already is still true, but first let us consider the alternative.

8.3.1 Physical Memory

A simple memory management system would use an address space that directly maps on to what it is trying to describe. This is analogous to most street addressing, where all the buildings on a given street will start at number 1 at one end, and then increment along the street. This is a physical mapping as number 1 refers to the first house, number 2 to the house next to it and so on.

The advantage of such a system is that it is extremely quick to derive and use. As soon as you're told and address, you know exactly where to go, or what data to read without any additional steps. The downside is that the simplicity makes it hard to enforce access to any particular bit. Again, using the street address analogy, consider that most military bases don't appear on road maps. This doesn't make it impossible to find them, but its suddenly a lot harder to send them a letter.

Such physically mapped systems are still used in some computing systems, especially in very small or simple systems that typically only perform a single function on very limited hardware. However, this is certainly not a common method on any system complex enough to be running an OS.

8.3.2 Virtual Mapping

Rather than directly mapping address to memory, the OS will provide each process with a virtual mapping. This is the same as in the physical example, where there is a collection of individual virtual addresses, which still map to some memory location. The process will treat these addresses as though they are a physical mapping, and so will still read and write anything to the addresses in that mapping, but the process has no idea what physical address each virtual address actually maps to.

Each process is given its own virtual memory space, which may or may not overlap. It doesn't really matter if they do or not as they are completely isolated, so what is address 1000 for one process, will almost certainly not map to the same location as address 1000 in another process. Exactly how much virtual memory is given to each system is configured on a per device basis, though this is usually related to how much RAM your device has.

8.3.3 Page tables

Within virtual memory mapping, address are stored in page tables which directly map virtual address to physical addresses. This address translation takes place in a dedicated piece of hardware called the Memory Management Unit (MMU). Each address will usually be for a memory page, rather than an individual byte. This is as memory is divided into pages (typically of 4KB) so that we don't need to store addresses of every individual byte. There is also no requirement that our virtual memory is actually next to each other in physical memory. Therefore virtual address 1 could point to one location in physical memory, while virtual address 2 points to some very very different location.

This demonstrates one advantage of virtual memory, as even in this example the programmer can navigate around this virtual memory in a linear fashion, and the fact that it is actually quite disparate in memory doesn't even appear to them.

8.3.4 Sizing Virtual Memory and Physical Memory

Just as there is a limit to how much space a physical mapping can map, there is a limited size to the virtual address space, though they might not be as linked as you would expect. You can absolutely have a bigger virtual memory space than physical, and this is in fact used in most systems. As the limit on your physical memory is usually your RAM, this is often taken as the starting size for your virtual memory. However, many systems use the concept of 'swap' memory, where we allow for a slightly larger virtual memory than physical memory. This means we can store more data, and are less likely to run into problems of loading programs too big to store.

However, we haven't actually expanded our memory, and though we can refer to more pages than really fit into our memory, we are going to need to swap pages in and out of cache if we are to use such a system. If we only add a small amount of swap memory (10%) then we aren't going to see too many cache misses, but if we add far too much then we're going to defeat the entire point of the cache as we'll just be swapping things in an out constantly. Usually locality will save us here and minimise misses, but again, too large a swap will sink us here if we aren't careful.

Chapter 9

Networks

Although there are many kinds of computer networks created, the vast majority of network equipment is based on the TCP/IP stack, which we will cover in these notes. These course notes focus on explaining how the network is implemented and the properties that arise from the implementation. Some reference material for network programming is included at the end of the chapter, but the exercises are intended as the main way to learn this.

9.1 OSI layers

The OSI layers are a model that explains the implementation of the internet. Each of the layers in the OSI model relies on the previous layer and makes certain assumptions about it. Using the services provided by the underlying layer, each layer can add new services. There are several different versions of the OSI model, with differing numbers of layers and various names for those layers. For this course we will only consider the commonly used 5 layer model shown below. Just make sure to remember that different representations do exist.

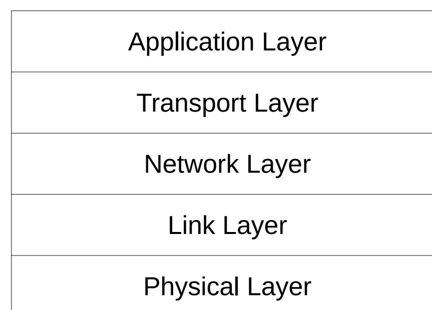


Figure 9.1: The 5 layer OSI model

By using a layered model we can seperate out functionality, to aid modelling, but also product design and deployment. By letting individual components (software or hardware) only need to meet the requirements of an individual layer, then companies or developers can more easilly create small components that can be of use to the wider network. This has greatly aided the very fast deployment of a global network of devices, which would not be possible if each user needed to implement an entire end to end system. The OSI model is intended to describe many kinds of networks, but in the text here we map the concepts to the implementation on global internet and consider this the only implementation.

9.2 Physical layer

The physical layer is the lowest layer in the OSI model and is concerned with transmission of bits via various propagation media. The first networks were implemented as wired networks, propagating electrical signals through a shared copper wire. Modern implementations use radio signals, to implement services such as Wi-Fi and 3/4/5G.

The physical layer provides the service of sending and receiving bits. It is important to note that due to properties of the propagation medium, the timings, bandwidth, and error correction properties are different. Each of the layers above needs to accept this, such that the layers work similarly with different physical layers.

9.2.1 Sharing a medium

The most significant work done by the physical layer is the ability to transmit messages on a shared medium, without any out-of-band control mechanism. The primary method for solving this is the use of a multiple-access protocol. For both (wired) Ethernet and Wi-Fi, the protocol is the carrier-sense multiple-access collision, CSMA, protocol. The protocol requires that each host can "sense" if data is being sent, and refrains from sending data which would cause a collision, making the data impossible to read. Since there is no out-of-band communication, collisions will eventually happen when two hosts communicate. The CSMA protocol uses a concept of exponential back-off with random starting values to ensure that collisions are eventually resolved.

9.2.2 Sharing with radio signals

For radio-based networks, it is possible to have a situation where the base station (the Wi-Fi access point) is placed between two hosts, such that neither can sense the other host's signals. In this case the CSMA protocol is extended to use collision avoidance, as the detection is not always possible. The concept is explored in this animated video: `https://www.youtube.com/watch?v=iKn0GzF5-IU`.

9.2.3 Connecting: Hub

Since the physical layer is about transmitting signals, and a shared media is supported, the physical layer allows multiple machines to be joined. For Ethernet (wired) networks, a hub can be used to physically connect multiple devices. With a network connected by a hub, the entire bandwidth for the network is shared with all hosts, resulting in poor scalability. If a network comprises 100 hosts and is using 100 Mbps Ethernet, each host will reach less than 1 Mbps if all hosts attempt to communicate at full speed.

9.3 Link layer

With a physical layer that is capable of transmitting bits, the link layer provides the option for addressing a single host. The bits are *framed* to allow sending a number of bits. Since the physical layer is expected to be using a shared medium, the link layer assumes that all frames are broadcast to every host.

9.3.1 Addressing network cards: MAC

To solve the problem of sending a frame to a particular host, the link layer adds *media access control*, MAC, addresses. Every network device has a globally unique address that is typically burned into the device but can be changed in some cases. When a host wishes to communicate with another host, a frame is transmitted on the physical medium, containing the MAC address of the recipient and sender.

The format of the MAC address is using six dual-digit hexadecimal numbers, for instance: `01:23:45:67:89:ab`. The special address `ff:ff:ff:ff:ff:ff` is used for broadcasting frames to all recipients.

9.3.2 Connecting: Switch

To make larger networks more efficient, a switch can be used in place of a hub. Where the hub operates on the physical layer, a switch operates on the link layer and forwards frames. When a switch is powered on, it has no knowledge of the network. Each frame it receives, it will broadcast to all other ports with a cable plugged in. For each package it receives, it will record the sender MAC address as well as the originating port. Since each MAC address is globally unique, the switch can use this simple scheme to learn where to send a frame and can avoid broadcasting to the entire network. Note that this feature works without any configuration, and even supports networks of networks, as the switch will allow mapping multiple MAC addresses to a single port. If a host is moved to another port, there will be a short period where the switch will forward frames incorrectly, until it sees a package from the new port.

9.3.3 Switches can increase available bandwidth

Once the network has levels, or just a large number of hosts, switches become a crucial component in ensuring performance of the network. More expensive switches can allow multiple parallel full-duplex data paths, such that multiple pairs of ports may use the full bandwidth. This allows the network to scale better with the addition of multiple hosts but may still have bottlenecks if there is cross-switch communication.

9.3.4 Transmission errors

Since the physical layer *may* have transmission errors, the link frames typically include a simple checksum that the receiver verifies. If the checksum is incorrect, or any parts of the frame are invalid, the frame is *dropped* with no notification to the sender.

9.4 Network layer

With a layer that supports sending frames, the network layer adds routing and global addressing. While we *can* build larger networks with a switch, the idea of broadcasting becomes problematic for a global network.

9.4.1 Addressing hosts: IP address

Because the MAC address is fixed, it is not usable for global routing. Imagine that a core router on the internet would need to keep a list of all MAC addresses for all machines in Europe. The size and maintenance of such a list would be an almost impossible task.

To simplify routing, each host on the network layer has at least one IP address, used to transmit *packages*. The IP addresses are assigned in a hierarchical manner, such that a geographic region has a large range, which is then divided into smaller and smaller ranges. At the bottom is the internet service provider, ISP, who owns one or more ranges. From these ranges they assign one or more IP addresses to their customers.

The hierarchy is important when considering global-scale routing. Instead of having a core router that keeps track of all IPs for Europe, it can just know that one of the ports "eventually leads to Europe" and then assign a few IP ranges to that port.

9.4.2 IP, CIDR, networks and masks

The IP addresses are 32 bit numbers, and usually presented as four 8-bit decimal numbers, called the dotted-decimal notation: `123.211.8.111`. The routing works locally by comparing one IP address with another, using bitwise `XOR`. By `XOR`'ing two IP addresses, any bits that are the same will be zero. The subnet mask then defines what bits are ignored, so we can use bitwise `AND`

on the result. After these two operations, the result will only be all zero bits if two IP addresses are on the same subnet.

As an example, consider the two IP addresses `192.168.0.8` and `192.168.0.37`. Given a subnet mask of `255.255.255.192`, we can apply the `XOR` operation to the two numbers and get `0.0.0.45`. When we apply the `AND` operation to the result we get `0.0.0.0`, meaning that the two hosts are on the same subnet.

As the subnet masks are always constructed with leading zeroes (i.e. it is not valid to use `255.0.255.0`), we can simply count the number of leading bits in the mask. This gives the classless interdomain routing notation, where we supply the IP address and number of bits in a compact notation: `192.168.0.0/26`. This notation is equivalent to supplying an IP address and a subnet mask, as we can convert trivially between the two.

9.4.3 Packets

So far we have mentioned packages a few times without really describing what they are. Most messages sent over the network are small (maybe only a couple of hundred bytes in length), but there are manay messages that considerably larger than this. If a file of several GB in size was sent continuously over the network it would take considerable amount of time, and no other communication could take place in that time. This is how traditional network communication such as old landline phones used to opperate and is has the additional inefficency of the space for the message on the network needs to be reserved, even if no message is actually being sent at that time. Modern networks split up messages into packets, typically of no more than 1500B in size. These packets can be interleaved with other messages being sent over the network to share the available resources. Packets may be recieved out of order at the recieving end of the communication, in which case the protocols discussed below will re-order them correctly before passing the message on to the application layer.

9.4.4 Connecting networks: Router

If we consider a package sent from a home-user in Europe to a server in the USA, the package will initially be sent to the *uplink* port of the routers, until it reaches a router that can cross the Atlantic ocean. After it has crossed the Atlantic ocean, it will reach more and more specific IP ranges until it arrives at the destination. This system allows each router to know only general directions, simplified as "one port for away, multiple closer". As mentioned in the video, routing is often done with "longest prefix matching", where multiple rules are stored as a tuple of: `(destination port, IP pattern in CIDR)`. The router can then sort the rules by the number of fixed bits (i.e. the `/x` part of the CIDR address), and use the first forwarding rule where the bits match. This approach allows the router to store broad "general rules" and then selectively change smaller ranges. As the only operations required are `XOR` + `AND`, it can be performed efficiently in hardware.

9.4.5 A router is a host

The router device itself works on the network layer, such that it can read the IP address. It can be considered a specialized computer, because the first routers were simply ordinary computers with multiple network cards. Unlike the switch and hub, a router is visible on the network and is addressable with its own IP addresses.

To function correctly, a router needs to know what ranges its ports are connected to, which requires manual configuration. As the network can also change, due to links being created and removed, as well as traffic changes and fluctuating transfer costs, the routers need to be dynamically updated.

9.4.6 Updating routing tables

The dynamic updates are handled inside each owners' network, and across the networks using various protocols, constantly measuring capabilities, traffic load, and costs. One of the protocols for communicating routes between network owners is called the Border Gateway Protocol and is unfortunately not secure yet. Bad actors can incorrectly advertise short and routes, which causes the networks to start sending all traffic over a particular link, either for disruption or eavesdropping purposes. Occasionally this also happens due to human errors, leaving parts of the internet unreachable for periods of time.

For an overview of the layers until now, and the different components that connect them, there is an animated video here: `https://www.youtube.com/watch?v=1z0ULvg_pW8`.

9.4.7 Packet loss

Each point on the network, be it routers or hosts, will maintain some cache for the storing of packets. The recieving of packets, writing them to cache, waiting for them to be transmitted, and the time taken to write them onto the network all conmtibute to a nodal delay, e.g. the delay in sending a packet at a given node. Nodes will have a finite sending speed and if packets are recieved faster than they can be sent then the local cache will fill up. If this state persists then eventually the cache will fill and packet loss will occur, where a packet is discarded or overwritten. Any information in this packet will be permanently lost.

9.5 Transport layer

With a network layer that is capable of transmitting a package from one host to another, the transport layer provides one protocol for doing just that: User Datagram Protocol, UDP.

9.5.1 Adding ports

A datagram in UDP adds only a single feature to the service provided by the network layer: ports. To allow multiple processes on a given host to use the network, UDP adds a 16-bit port number. When describing and address on the transport layer, the port number is often added to the IP address or hostname after a colon: `192.168.0.1:456`. The operating system kernel will use the IP address and port number to deliver packages to a particular process (More on this below). However, since UDP uses the network layer, there is no acknowledgement of receipt and no signals if a package is lost. Because the routers update dynamically, it is also possible for packages to reach the destination via different routes, causing packages to arrive in a different order than they were sent.

UDP does not make any attempt to detect packet loss, and so it is up to an Application to determine when to resend data or not. Despite this, UDP can be acceptable for some cases such as game updates or video streams, where we would rather loose a frame than have a stuttering video. But for many other cases, such as transferring a file or dataset, the UDP service is not useful.

Note that port numbers can be anything up to 65000, but that it is common practice to avoid 0-2000ish as these are reserved for 'well-known' services. For example, most websites are hosted on port 80, and to keep functionality clear we politely avoid using 80 for non-website applications.

9.5.2 Transmission control protocol: TCP

The Transmission Control Protocol, TCP, is the most widely used protocol and most often used with IP to form TCP/IP. The TCP protocol builds a reliable transfer stream on top of an unreliable delivery provided by the network layer. The protocol itself is very robust, with understandable mechanics, but can require some trials to accept that it works in all cases.

9.5.3 Establishing a connection

Since the network layer does not tell us if a package has been delivered, the TCP protocol uses *acknowledgements*, ACKs, to report receipt of a package. Before a connection is establish, the client sends a special SYN message, and awaits an ACK, and sends an ACK. This exchange happens before any actual data is transmitted but allows for "piggy backing" data on the last ACK message. When designing an application, it is important to know that this adds an overhead for each established connection, and thus connections should preferably be reused where possible.

9.5.4 A simple stop-and-go TCP-like protocol

Once the connection is established, data can flow, but due to the network layers unreliability, we can receive packages out-of-order or lose them. For a simple protocol, we can just drop out-of-order packages, treating them as lost. The

remaining problem has two cases: loss of package and loss of ACK. Since the sender cannot know which of these has occurred, it assumes the data is lost, and re-transmits it after a timeout has occurred while waiting for an ACK. If we prematurely hit a timeout, we will re-send a received package, but this is the same case as a lost ACK will produce.

The recipient can simply discard a package it has already received, so if we add a package number, called a sequence number[1], to the package, it is trivial for the recipient to know which packages are new and which are retransmits. It should be fairly simple to convince oneself that this works in all cases, in the sense that the recipient will eventually get the package, and the sender will eventually get an ACK.

9.5.5 Improving bandwidth with latency hiding

However, due to the communication delay, we are waiting some of the time, instead of communicating. For even moderate communication delays, this results in poor utilization. To work around this, the TCP protocol allows multiple packages to be "in-flight", so the communication can occur with full bandwidth. This requires that the sender needs to keep track of which packages have gotten an ACK and which are still pending. We also need to keep copies of multiple packages, such that we can retransmit them if required. This does not change the way the simple stop-n-go protocol works, it simply adds a counter on the sender side. The choice of "how many" is done with a ramp-up process, increasing until no improvements are seen, which further adds to the delay of new connections.

9.5.6 Minimizing retransmission

We could stop there, and be happy that it works, but if we get packages out-of-order it means not being able to send the ACK, and many in-flight packages needs to be retransmitted. This is solved in TCP by keeping a receive buffer, allowing packages to arrive in out-of-order. This does not change the protocol, except that the recipient needs to send an ACK, only when there are no "holes" in the sequence of received packages. This is simplified a bit in TCP, where an ACK is interpreted as vouching for all data up until the sequence number. This means that when as the sender we recieve an acknowledgetment for packet 500, we know that all packets up to 500 have also been recieved. This means that lost ACK messages are usually not causing disturbance, as a new one arrives shortly afterwards. But it also makes it easier for the recipient to just send an ACK for the full no-holes sequence.

If we lose a package, the recipient will notice that it keeps getting packages with the same sequence number. Since the reciever can only ACK the last packet it has recieved, it will keep doing so. The sender can then notice getting

[1] In the text and the video, we use a number pr. package. In the real TCP protocol, the sequence number counts bytes to support split packages.

multiple ACKs (specifically: 3) for the same package, and guess that it needs to re-transmit the next package. This improvement makes it possible to transmit at full speed even with some package loss, and without having to eait for time-consuming timeouts.

Hopefully you can convince yourself that the protocol works correctly in all cases, despite the performance enhancements.

9.5.7 Flow control

When designing and evaluating network performance, it is important to know the two complementary mechanisms that both end up throttling the sending of packages. The original throttling mechanism is called flow-control and works by having the recipient include the size of the receive buffer with each ACK message. The sender can monitor this value and reduce the sending rate if it notices that the remaining space is decreasing. This is essentially trying to prevent the sender sending packets that there is not space to recieve, preventing packet loss and package re-transmission. Once the space in the reciever is zero, the sender will wait for a timeout or an ACK with a non-zero receive buffer size. The timeout is a protection against the case where the ACK is lost.

9.5.8 Congestion control

The other mechanism is congestion control, which is a built-in protection against overflowing the network itself. While any one machine cannot hope to overflow the core routers in the network, many hosts working together can. What happened in the early days of the internet was that some routers were overwhelmed and started dropping packages. The hosts were using TCP and responded by retransmitting the lost packages, causing a build-up of lost packages, to the point where no connection was working. The TCP protocols are implemented in software, so the TCP implementations were gradually updated with congestion control additions. Unlike flow-control, there is no simple way of reporting the current load of all routers in the path. Instead, congestion control monitors the responses from the client, and makes a guess of the network state. Various implementations use different metrics, where the loss of packages was once seen as the right indicator. However, since package loss occurs *after* the congestion issues have started, this was later changed to measure the time between ACK messages. Once the routers start to get overloaded, the response time increases, and modern TCP implementations use this to throttle the sending speed.

9.5.9 Closing a connection

The layers on top of TCP can assume that packages have arrived and are delivered in-order. But how do we know if we have received all packages, and not lost the last one? The TCP implementation handles this by sending a package with the FIN bit set. Like other packages, the FIN package can be

lost, so we need to get an ACK as well. Again, the ACK package may be lost, so we need another package, and so on. TCP has a pragmatic solution, where the side that wishes to close the connection will send FIN, wait for an ACK and then send a final ACK. The final ACK *may* be lost, in which case the recipient does not know if the sender has received the ACK. If this happens, TCP will keep the connection in a "linger" state for a period before using a timeout and closing the connection. Even if this happens, both sides can be certain that all messages have been exchanged.

9.5.10 Network Address Translation

In the original vision of the internet, all hosts were publicly addressable with their own IP address. Later that turned out to be a bad idea for security and economic reasons. Each ISP has to purchase ranges of IP addresses and assign these to their customers. But some customers may have several devices, increasing the cost. Likewise, some customers and companies may like to have an internal network with printers and servers which is not exposed to the internet, but at the same time be able to access the internet.

The technique that was employed rely on the router being a computer itself, with an internal and external IP address. When a host sends a package destined for the external network, the router will pick a random unused port number and forward the package, using the routers external IP address and the randomly chosen port. This information is stored in a table inside the router, such that when a response package arrives that is destined for the particular port and external IP, the router will forward it to the internal network, using the original IP and port.

This operation is transparent to the hosts inside and outside the network, allowing ISP customers to have multiple internal hosts, sharing a single external IP address.

When compared to the original vision for the internet, the NAT approach breaks with the assumption that each host needs a unique IP. In practice this means that a NAT'ed host can only initiate connections, it cannot receive new connections (i.e., it cannot be the server, only the client).

In some cases, it might be desirable to use NAT, but still have a host act as a server. For this situation, most NAT capable routers allow pre-loading of static rules, for instance "external IP, port 80" should go to "10.0.0.4:1234". This is the same operation that happens automatically for outbound connections, but just always active.

Although NAT is not strictly a security feature, it does provide some protection against insecurely configured machines being directly accessible from the internet. That is unless your NAT capable router has Universal Plug-n-Play, UPNP, which allows any program on your machine to request insert a preloaded NAT rule, exposing everything from printers to webcams without the owners knowledge.

9.5.11 IPv6

In the above we have only considered IPv4, which is where the addresses are 32-bit. Despite the visions of giving every host their own IP address, the number of hosts in the world has long exceeded the available IPv4 numbers. Thanks mostly to NAT techniques, and a similar concept for ISPs called carrier-grade-NAT, this has not yet stopped the growth of the internet.

Before it became apparent that NAT would ease some of the growing pains, the IPv6 standard was accepted and ratified. With IPv6 there are now 128 bits for an address, essentially allowing so many hosts, that it is unnecessary to have ports or NAT anywhere.

Where the transport layer is implemented in software and easily updated, the network layer is embedded in devices with special-purpose chips. Changing these is costly and has so far dragged out for more than a decade.

The number of IPv6 enabled hosts and routers continue to increase, but there are still many devices with a physical chip that cannot upgrade to IPv4. Many of these are IoT devices, attached to an expensive TV, surveillance cam or refrigerator. As the devices work fine for the owners, there is virtually no incentives for replacing it, leaving us with a hybrid IPv4 and IPv6 network for some time to come.

New internet services would likely strive to use IPv4 addresses as there are plenty of ISP that only offer IPv4. If a company only has IPv6 servers, they would be inaccessible to a number of potential customers.

Currently the most promising upgrade seems to be, once again, relying on NAT to perform transparent IPv4 to IPv6 translations. This could allow the internet as a whole to switch to IPv6 while also being accessible to IPv4 users.

9.6 Application layers

With the transport layer providing in-order delivery guarantees for communicating between to processes, it opens up to a multitude of applications. The most popular one being the use of HTTP for serving web pages, but also many other services, including the network time protocol, which keeps your computer clock running accurately even though it is has low precision.

9.6.1 Domain Name System

One application that runs on top of the transport layer is the domain name system, DNS. It is essentially a global key-value store for organizing values belonging to a given domain name. In the simplest case it is responsible for mapping a name, such as `google.com` to an IP address. This makes it easier for humans to remember, but also allows the owner of the domain name to change which host IPs are returned, without needing to contact the users. This might occur for a variety of reasons, hardware may simply be changed, or a different content provider may be used depending on your physical location. After all, if you are in Australia you could still use `cnn.com` to access a news

site, but a local server would be returned rather than connecting all the way to New York.

The design of the system is based on a hierarchical structure, where 13 logical servers replicate the root information. In practice these 13 servers are implemented on over 700 machines, geographically distributed over the entire globe.

The root nodes store the IP addresses of the top-level domain servers, where a top-level domain could be `.com`. A top-level domain server, naturally replicated on multiple hosts, keeps a list of the name servers for each domain ending with the top-level domain (i.e. the top-level server for `.com` keeps track of `twitter.com`, `google.com`, etc).

This means that each domain must run their own nameserver, but in practice, most nameservers are run by a set of registrars, where a small number of machines are responsible for thousands of domains.

The domain name server is the final step, containing all information related to a given domain name. This server can be queried to obtain IP addresses for all subdomains (i.e., `www.google.com`, `docs.google.com`) as well as the domain itself (i.e., `google.com`).

Apart from the IP addresses for hosts, the DNS records contain email servers, known as `MX` records, free text, called `TXT` records, and other domain related information.

To keep the load on DNS servers down, each host in the DNS system will cache the values it receives for a period. The exact period is also stored in the DNS records with a time-to-live, `TTL`, value expressed in seconds.

This distributed system means that no device needs to maintain a complete record of the entire DNS heirarchy, which would be a considerable task in todays internet. It also means that organisations can maintain control over their own DNS records. For instance, KU will maintain a DNS server which it can administer, adding or removing web-pages and resources as it sees fit. This does not need to be registered with the wider internet, only the location of the KU DNS server needs to be known. Any requests for any IP addresses can just be sent to this domain name server and it will handle it, however KU wishes to do so. Another advantage of such a system is that there is no single point of failure, in fact that are tens of thousands of DNS servers throughout the world, making it infeasible for anyone to bring down the entire system.

9.6.2 DNS caching

When it comes time for a user to make a DNS querry, they will send a request to a local DNS server. This will then querry the root server for the corresponding top-level domain server. The top-level domain server is then contacted by the local DNS server to obtain the address of the authorative DNS server. The local DNS server will then contact this authorative server for the final DNS address. This structure is refered to as iterative, as it is run as a series of independent querries by the local DNS server. In contrast to this is the recursive querry, where the local DNS server will contact the root, with the root then contacting

the appropriate top-level domain server which in turn contacts the authorative server. Though this may seem very similar, in practice it makes much better use of caching. DNS requests are frequent, small, and very often predictable (how often do you really go to a site you've never been to before). For these reasons DNS is a very cachable application, to save repeated requests for the same information. Recursive makes better use of caching as a cache hit at any stage of the process will save any subsequent messages, but in the iterative DNS only a cache hit at the local DNS would save messages.

9.7 Network Programming

These brief notes will only look into using Python for network programming. All network programming is done via sockets, which act as file descriptors we can read and write to/from. A socket will map to a port via the `bind` function. From here we can either use it to listen for incoming messages via the `listen` function, or attempt to establish a new connection to another port. Note that the `connect` function includes the bind function within it so we don't need to call it explictly. An example is shown below. This creates a socket, connects to some other host using an IP and port number, assembles a message, and sends that message.

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) \
    as client_socket:
  client_socket.connect(("127.0.0.1", 12345))
  request = bytearray("message".encode())
  client_socket.sendall(request)
```

Receiving a message can also be done through sockets. This example creates a socket at a given IP and port, listens for inbound communications, accepts those connections, reads the request and replies.

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) \
    as server_socket:
  server_socket.bind(("127.0.0.1", 23456))
  server_socket.listen()

  while True:
    connection, connection_address = server_socket.accept()
    with connection:
      message = connection.recv(1024)
      connection.sendall(response)
```

Though individual socket programming may be usefull at some points, especially for sending messages. Servers are usually implemented through the

socketserver class as shown below. In this model, dedicated handlers for requests are created so that responses can be processed concurrently. This concurrency is automatically handled by the `ThreadingTCPServer` class, which wil create a new thread for each inbound communication.

```
from socketserver import ThreadingTCPServer, \
        StreamRequestHandler

class MyHandler(StreamRequestHandler):
        def handle(self) -> None:
                message = self.request.recv(1024)
                self.request.sendall(message)

with ThreadingTCPServer(("127.0.0.1", 5678), MyHandler) \
                as my_server:
        my_server.serve_forever()
```

Recall that regardless of the method used to program the sockets, all communication should be conducted via a client-server model. Under this model a communication is initiated by the client, aka the sender. It will attempt to contact a server, or reciever. Depending on the protocol, the server may respond or not. It is common that a response is always expected, such as in most TCP communications. Even if a nonsense message is sent by the client, the server will respond with an error message. Under the client/server model, if a response is expected, the client must commit to being able to recieve a reply, and the server must commit to sending a reply in a finite amount of time. This ensures the system won't livelock, where the communication hangs without completing. Note that hosts can act as both clients and servers at the same time, e.g., by recieving a request which requires some network communication to respond to. This is fine, but within each communication, one host will always be the client, and one will be the server.

A server will create a socket to listen for new connections as shown above. This will be on a defined port, however, this is not the port that the connection will actually be established on. If this were, then the connection would reserve that port and no other clients could connect. Therefore, when a server recieves a new connection request, it will use the `accept` function to create and bind a new socket for the inbound communication. This makes servers capable of responding to multple connections, though the handling will have to be threaded in order be responded to concurrently. This will be investigated further in the following Chapter, though as already noted, the concurrency is automatically handled in the socketserver implementation.

Chapter 10

OpenMP

Writing multi-threaded programs using the raw POSIX threads API is tedious and error-prone. It is also not portable, as POSIX threads is specific to Unix systems. Also, writing efficient multi-threaded code is difficult, as thread creation is relatively expensive, so we should ideally write our programs to have a fixed number of *worker threads* that are kept running in the background, and periodically assigned work by a scheduler. In many cases, particularly within scientific computing, we do not need the flexibility and low-level control of POSIX threads. We mostly wish to parallelise loops with iterations that are *independent*, meaning that they can be executed in any order without changing the result. For such programs we can use *OpenMP*. We will use only a small subset of OpenMP in this course.

10.1 Basic use of OpenMP

OpenMP is an extension to the C programming language[1] that allows the programmer to insert high-level *directives* that indicate when and how loops should be executed in parallel. The compiler and runtime system then takes care of low-level thread management. OpenMP uses the the *fork-join* model of parallel execution: a program starts with a single thread, the *master thread*, which runs sequentially until the first *parallel region* (such as a parallel loop) is encountered. At this point, the master thread creates a group of threads ("fork"[2]), which execute the loop. The master thread waits for all of them to finish ("join"), and then continues on sequentially. This is called an *implicit barrier*: a point where execution pauses until all threads reach it. For example:

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
  A[i] = A[i]*2;
}
```

[1] OpenMP is not C specific, and is also in wide use for Fortran.

[2] Note an unfortunate mix-up of nomenclature: this has nothing to do with the Unix notion of `fork()`, which creates *processes*, not *threads*

The **#pragma** omp parallel **for** line is an OpenMP directive that indicates that the iterations of the following `for` loop can be executed in parallel. If this loop is compiled with a compiler that supports OpenMP, the `n` iterations of the loop will be divided among some worker threads, which will then execute them in parallel.

The number of threads used can be controlled at run-time, and is usually not equal to the number of iterations in the parallel loop. This is because when the amount of work per iteration is small (as above), it would not be efficient to have one thread per iteration.

One important idea behind OpenMP is that to understand the semantics of a program, we can always remove the directives and consider what the remaining sequential C program would compute. This is called the *sequential elision*. This is a great advantage over low-level multi-threaded programming.

When we ask OpenMP to parallelise a loop, we solemnly swear that the following **for** loop is actually parallel. If we break this vow then the parallel and sequential execution of the code will give different results; the API does not provide any guarantees about the absence of race-conditions. For example, the iterations of the following loop are not independent, yet OpenMP will not stop us from asking it to be executed in parallel:

```
sum = 0;
#pragma omp parallel for
for (int i=0; i<N; i++) {
  sum += A[i];
}
```

Since all threads executing the parallel region have access to the same data, it is easy to have accidental race conditions in OpenMP programs. In chapter 12 we will look in detail at determining when it is safe to execute a loop in parallel, and how to transform loops so they become safe to execute in parallel.

10.1.1 Compiling and running OpenMP programs

To compile with support for OpenMP directives in `gcc`, pass the `-fopenmp` option to the compiler. The number of threads that are going to be used for parallel execution can be set by environment variable `OMP_NUM_THREADS`. For example,

```
$ export OMP_NUM_THREADS=8
```

sets the environment variable for the current shell session, such that any OpenMP we run will use eight threads. Determining the optimal number of threads to use for a given program on a particular machine is something of a black art. In practice, we just try a few different numbers and see what runs fastest.

For example, suppose the contrived program in listing 10.1 is stored in the file `openmp-example.c`. We can then compile as follows:

Listing 10.1: A very simple example of using OpenMP.

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

int main(void) {
  int n = 100000000;

  int *arr = malloc(n*sizeof(int));

#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    arr[i] = i;
  }

  free(arr);
}
```

```
$ gcc -o openmp-example openmp-example.c -fopenmp
```

And then run with various values of `OMP_NUM_THREADS` to investigate the impact of parallelisation:

```
$ time OMP_NUM_THREADS=1  ./openmp-example
real    0m0.124s
user    0m0.034s
sys     0m0.090s
$ time OMP_NUM_THREADS=2  ./openmp-example
real    0m0.076s
user    0m0.033s
sys     0m0.104s
$ time OMP_NUM_THREADS=4  ./openmp-example
real    0m0.054s
user    0m0.039s
sys     0m0.133s
$ time OMP_NUM_THREADS=8  ./openmp-example
real    0m0.046s
user    0m0.054s
sys     0m0.184s
```

Note how the *real* time drops as we use more threads—although it's not quite eight times as fast with eight threads as with one. This is likely because this contrived program does so little work compared to the amount of memory we are accessing.

10.1.2 Parallelism versus Concurrency

The terms *parallelism* and *concurrency* are frequently and historically used interchangeably. If you look then up in a dictionary, you will find them to have almost the same definitions. In computer science, they are terms of art with distinct (although related) meanings.

To illustrate concurrency, consider a video game, which from a programming perspective is basically a real-time interactive simulation. Many things need to happen concurrently:

- We need to figure out what sounds and music to play and send it to the IO device connected to the speakers.
- Many times per second, we need to draw to the screen a rendering of the world as observed by the player.
- Perhaps we wish to guess at where the player is headed next, and preload those parts of the game world.
- We need to run artificial intelligence for computer-controlled enemies.
- We need to compute physics interactions.
- In a multiplayer game, we need to transmit information information about the local game world to other players across the network, as well as incorporate information we receive in return.
- Probably we also wish to perform cleanup tasks, such as removing objects from the game world that after a while are no longer necessary (e.g. the remains of deceased enemies), to clear up system resources.

From a programming perspective, it is nicer if we can write each of these parts as separate flows of control. The artificial intelligence code should not worry about constantly checking whether it's time to draw a new screen frame, or whether the player hit some key. Similarly, the rate at which we receive information from the network is completely unpredictable, relative to our other responsibilities.

One solution is to implement each of these parts as a distinct thread, and depend on the operating system to *context-switch* between them as needed. Maybe we can also use some form of *scheduling policy* that understands that it's more important for the threads responsible for music and graphics to run when they need to, than the thread that cleans up dead objects or simulates physics. If we have only a single processor, then all these different threads run *concurrently*, in that they overlap in time, but only one will physically be executing instructions at any given point in time. This means that concurrency can be a useful programming model even when the goal is not to make the program faster. For that matter, threads are not the only way to implement concurrency—asynchronous event loops are a popular technique for highly scalable web servers, but outside the scope of this course.

Listing 10.2: OpenMP dot product with reduction clause.

```
double dotprod(int n, double *x, double *y) {
  double sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < n; i++) {
    sum += x[i] * y[i];
  }
  return sum;
}
```

Definition 10.1 (Concurrency) *Concurrency is the use of multiple, possibly interacting, logical flows of control.*

Parallelism is about making programs faster by performing several computations at the same time. When we have a program with multiple threads, such as the video game example above, then if we have a machine with more than one processing core (which is essentially every machine these days), we can run several of those threads in parallel. While in Unix, threads are the fundamental *implementation mechanism* for obtaining parallelism, we often program in languages or frameworks that do not directly expose threads, because they can be difficult to work with. For example, OpenMP is a parallel programming model, but for simple parallel loops, we do not concern ourselves with actual threads, and we only have a single logical control flow.

Definition 10.2 (Parallelism) *Parallelism is the simultaneous use of multiple processing units, with the goal of speeding up a computation.*

In scientific computing we are mostly concerned with parallelising loops with independent iterations, and not with concurrent flows of control. While OpenMP allows us to peek beneath the covers and interact with the threads that it uses to *implement* the parallel loop abstraction, there does exist forms of parallelism, and parallel programming languages, that do not expose this abstraction, and are truly parallel without exposing any concurrency.

10.2 Reductions

A loop whose iterations are completely independent can be parallelised with the `#pragma omp parallel for` directive, as shown before. Another common case is when the iterations are *almost* independent, but they all update a single accumulator - for example, when summing the elements of an array. For such loops, OpenMP provides *reduction clauses*, as used to compute a dot product on listing 10.2.

Note that all iterations of the loop update the same `sum` variable. The reduction clause `reduction(+:sum)` that we added to the OpenMP directive

indicates that this update is done with the + operator. The compiler will transform this loop such that each thread gets its own *private* copy of `sum`, which they then update independently. At the end, these per-thread results are then combined to obtain the final result.

Reduction clauses only immediately work with a small set of built-in binary operators: `+`, `*`, `&&`, `||`, `&`, `|`, `^`, `max`, and `min`. It is also possible to use user-defined functions, but this is beyond the scope of this text. The common property shared by these operators is that they are *associative*, and have a *neutral element*. By associativity, we mean that for some operator $\oplus$, we have

$$(x \oplus y) \oplus z = x \oplus (y \oplus z).$$

That is, the "order of evaluation" does not matter. This is what allows us to partition the iterations of a reduction loop between multiple threads, without changing the result. Note that subtraction and division is *not* associative—this is we don't use them in reduction clauses.

A neutral element $0_\oplus$ for some operator $\oplus$ is a "natural zero" that does not change the result of evaluation:

$$x \oplus 0_\oplus = 0_\oplus \oplus x = x$$

For example, if the operator is addition, then the neutral element is 0. If the operator is multiplication, then the neutral element is 1. OpenMP requires that the initial value of each reduction variable (`sum` for listing 10.2) is the neutral element of the operator.

10.2.1 Associativity of floating point operations

Some of you might recall that addition and multiplication of floating-point numbers is *not* associative, due to roundoff errors. Yet we perform a reduction on **`double`** values in listing 10.2! How can that be valid? The short answer is that OpenMP allows us to shoot ourselves in the foot if we wish. In practice, floating-point operations are often *almost* associative, and we can get useful results by treating them as if they were. In particular, there is no reason to believe that a sequential left-to-right summation of floating-point numbers is going to be more numerically accurate than a parallelisation of that loop. This does mean that an OpenMP program that uses reductions on floating-point values might compute a different result than the original sequential program.

10.3 Nested loops

We can prefix every parallelisable **`for`**-loop with an OpenMP parallelisation directive. But what happens if we nest multiple parallel loops such as the following?

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
```

```
#pragma omp parallel for
  for (int j = 0; j < m; j++) {
    ...
  }
}
```

In principle, the answer is easy: the original master thread launches worker threads to handle each of the `n` outer iterations, and these individual worker threads may then launch more worker threads to handle the inner `m` iterations. This is called *nested parallelism*: iterations of a parallel loop may itself contain more parallel loops. The overhead of nested parallelism quickly becomes significant, in particular if we have recursive functions so that the nesting is *dynamic*, so in practice it is not widely used in OpenMP. In fact, OpenMP implementations are likely to ignore nested parallelisation directives, unless the `OMP_NESTED` environment variable is set to `True` when running the program.

However, a common case of nested parallelism is iterating across all elements of a multidimensional array, such as above, where we are conceptually covering all indexes of an `n` by `m` array. This is called *regular nested parallelism*, because the iteration count of the inner loop (`m`) is *invariant* (the same) for all iterations of the outer loop. Such a nested loop can be collapsed to a single loop that performs `n*m` iterations:

```
#pragma omp parallel for
for (int ij = 0; ij < n*m; ij++) {
  int i = ij / m;
  int j = ij % m;
  ...
}
```

We use division and modulo operations to extract the intended indexes from the "combined" index `ij`—this is actually the inverse of the row-major two-dimensional index function.

However, writing code like this is not very nice, as it obscures our intent. Fortunately, OpenMP provides the `collapse` clause that we can use to tell OpenMP to parallelise multiple *perfectly nested* loops. By perfectly nested, we mean that the outermost loops contain only a loop, and no other statements. An example is shown in listing 10.3, where we tell OpenMP to treat the two-deep *loop nest* as a single parallel loop.

You should generally use the `collapse` clause when writing such perfectly nested loops, which occurs frequently when implementing matrix operations. But more generally, it is usually not worth worrying too much about parallelising all inner loops of an OpenMP program. All we have to do is provide enough parallel work such that all processors on the system have work to do, and as of this writing, even a very large computer is unlikely to have more than 256 CPU cores—and on a personal computer, 16 is more likely.

Listing 10.3: Matrix addition with OpenMP.

```
void matadd(int n, int m,
            const double *x, const double *y,
            double *out) {
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      out[i*n+j] = x[i*n+j] + y[i*n+j];
    }
  }
}
```

10.4 Scheduling

When we use a directive to ask OpenMP to execute a loop in parallel, the compiler and runtime system will decide how the iterations should be distributed among the threads. By default, OpenMP uses *static scheduling*.

Definition 10.3 (Static scheduling) *When entering a parallel loop, we assign each thread exactly the number of iterations to execute.*

For example, when executing a loop with n iterations on m threads, we might assign $\frac{n}{m}$ iterations to each thread.

Static scheduling can be non-optimal when a loop is not *load-balanced*, meaning that not all loop iterations take the same time. For such an *imbalanced* loop, some threads may finish their iterations quickly and then sit idle while the other threads finish theirs. In such cases, we can ask OpenMP to use *dynamic scheduling*.

Definition 10.4 (Dynamic scheduling) *When entering a parallel loop, we assign each thread an iteration. When a thread finishes an iteration, it receives a new one to execute.*

The advantage of dynamic scheduling is that an idle thread will receive more work (if any is available). The disadvantage is that dynamic scheduling requires additional communication and synchronisation—while this is done for us by the runtime system, it carries a performance overhead.

As an example of the benefit of dynamic scheduling, consider parallelising loops where each iteration computes Fibonacci numbers using the recursive function defined in listing 10.4[3].

Since computation of $fib(i+1)$ takes over twice the time of $fib(i)$, we can produce a very imbalanced loop by letting iteration i compute $fib(i)$. Listing 10.5 shows how to parallelise this with a static schedule in OpenMP. On

[3]This is an inefficient way to compute Fibonacci numbers—we only use as an expensive computation that will take some time.

Listing 10.4: Recursive Fibonacci function.

```
int fib(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Listing 10.5: Fibonacci loop with static scheduling.

```
#pragma omp parallel for schedule(static)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

Listing 10.6: Fibonacci loop with dynamic scheduling.

```
#pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

my machine, for `n=45`, this program runs in $5.2s$. We can ask for dynamic scheduling by using the the `schedule(dynamic)` clause, as shown on listing 10.6. On my machine, this version runs in $2.27s$ - a speedup (section 11.1) of $2.29\times$.

By default, dynamic scheduling assigns single loop iterations to threads. This is not a problem for the Fibonacci example, because there are few loop iterations, and they take a fairly long time to run. For loops with many iterations, where dynamically scheduling single iterations at a time would involve too much overhead, we can add a *chunk size* to the scheduling clause. For example,

```
#pragma omp parallel for schedule(dynamic, 100)
```

will schedule in chunks of 100 iterations at a time.

The `guided` schedule is similar to `dynamic`, but starts out with big chunks and may then decrease the chunk size during program run-time if the work is imbalanced.

OpenMP's default behaviour will usually do a good job scheduling most loops. But if we do not see the performance gains we would expect, and we see in a process monitor that some of our processors are idle while our program

is running, it is worth considering whether our loops could benefit from a scheduling clause.

10.5 The Big Concurrency Problems

Concurrency and Parallel programming is a complex task, and can often be a study in what not to do. Luckily, OpenMP manages a lot of the underlying system interactions to ensure correct and efficient concurrency, but no system is perfect. If you ever need to debug whats going on, or try to implement your own concurrent solution you will need a firm understanding of the two core problems; race conditions and deadlocks.

10.5.1 Races

Definition 10.5 (Race Condition) *Sometimes also referred to simply as a race. A race condition is any processing whose final result depends on the arbitrary ordering of prior operations.*

Races are a result of concurrent programs least desirable feature, non-determinism. This is due to the essentially random[4] scheduling of any threads and processes, so we do not actually know in what order concurrent operations will be performed.

Within threading this is most often caused by global variables being shared among several threads. This is as each thread can read and write to and from the same location, in essentially any order. As an example, consider the code in listing 10.7. This code will spread the execution of the counting loop across many threads, which will all read, increment and write to count in an arbitrary order. This will produce an output for count that could be any value between 2 up to 1000000. In practice you are *very* unlikely to get a result as low as 2, but it is technically possible if you get the very unluckiest scheduling[5]. Of course its just as possible that you get 1000000 but this non-deterministic result is everything we want to avoid in computing.

10.5.2 Locks

The code in 10.7 will produce a non-deterministic result, which we have already seen how to solve in OpenMP, by adding 'reduction(+:count)' after the 'parallel for' loop. Reductions are great for OpenMP, but a more broadly applicable solution is the use of a mutex. Note that within the wider literature you will see references to mutexes, locks, or semaphores. These are all name for related but

[4]Note that in practice the scheduler will not be literally scheduling at random, but that process scheduling is way out of scope for this course. As user of a computer, we also have little to no control over other users and processes that our program may be competing with, so even when we do understand the scheduler it is still treated as effectively random.

[5]If you're interested in how this is possible you can look through the trace on page 36 of this pdf

Listing 10.7: An example of a racing program.

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

int main(int argc, char* argv[]) {

    int count = 0;
    int total = 1000000;
    #pragma omp parallel for
    for (int i=0; i<total; i++)
    {
        count++;
    }
    printf("Final count is: %d\n", count);
    printf("Should be:      %d\n", total);
}
```

slightly different objects but within HPPS, and we will only concern ourselves with mutexes. They are effectively a flag that can only be set by one thread at a time.

This is achieved via atomic operations. An atomic operation is one that cannot be interrupted. Recall that the problem with race conditions is that we cannot guarantee that a thread will not interrupt another thread, even if it is midway through an operation. Mutexes are implemented at the machine level to only have two operations, set (e.g. claim the mutex) and release (e.g. give up the mutex). Both of these operations are implemented as atomic operations, that the scheduler cannot interrupt. Once a thread has claimed a mutex, then any other threads that try to claim it will be blocked until the mutex is released. This can have the effect of reducing how much parallel processing is taking place, as threads will have to wait for mutexes to be released before they can continue. However, this cost to speed is worth it if it means we can get an actually meaningful result.

An example of how a mutex can be implemented using the 'pthread' library, as is shown in listing 10.8. This example will always produce the correct result, regardless of how many threads are used to calculate it. Note that in this case, the mutex is locking access to the entirety of the parallelised section, meaning that this program will in fact be much slower than the racing program shown in listing 10.7. We can see this in the timings presented below, when the mutexed version is 3 times slower, as the threads need to keep waiting for each other to release the mutex. This shows us that mutex usage should be minimised, by only locking the smallest number of instructions to ensure we get the correct result. Sometimes a lock is inevitable though, and so if we were going to improve this program we would perhaps restructure the entire code so that

Listing 10.8: An example of a mutex fixing a racing program.

```
#include <pthread.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {

    int count = 0;
    int total = 1000000;
    pthread_mutex_t lock;

    pthread_mutex_init(&lock, NULL);

    #pragma omp parallel for
    for (int i=0; i<total; i++)
    {
        pthread_mutex_lock(&lock);
        count++;
        pthread_mutex_unlock(&lock);
    }

    printf("Final␣count␣is:␣%d\n", count);
    printf("Should␣be:␣␣␣␣␣␣%d\n", total);
}
```

each thread calculates a subtotal, and these are then totalled in a mutexed variable. This is in fact what an OpenMP reduction automatically does.

```
$ time ./race
real    0m0.031s
user    0m0.173s
sys     0m0.004s
$ time ./mutex
real    0m0.102s
user    0m0.161s
sys     0m0.484s
```

10.5.3 Deadlock

Mutexes may help solve the problem of races, but they can introduce a compeletely new problem, deadlock.

Definition 10.6 (Deadlock) *Any situation where no system progress can take place, as every process/thread is waiting for another to progress before it can.*

Much like races, deadlocks can be non deterministic which can make them twice as annoying to debug, but we must treat the possibility of a deadlock as though it will deadlock eventually. Therefore we need to design our systems so that they are deadlock free. As an example of how a deadlock could occur consider the following two threads, each which just lock and then unlock two mutexes each:

```
Thread 1          Thread 2
 lock(A)           lock(B)
 lock(B)           lock(A)
unlock(A)         unlock(A)
unlock(B)         unlock(B)
```

The ordering of these operations is completely up to the whims of the scheduler. Many orderings of these operations would be fine, but one such bad one would be if thread 1 locked mutex A, but was then immediately interrupted by thread 2 who would then lock mutex B. Thread 2 cannot continue as it cannot lock mutex A, as it is already locked by thread 1. Thread 1 also cannot continue as its next operations is to lock mutex B, but it is locked by thread 2. This is a deadlock, as there is no way for either thread to progress. There is also no way for these two threads to detect that they are in a deadlock, the system is completely stopped with no way to recover.

10.5.4 Progress Graphs

Reasoning about how each thread could be scheduled is all well and good, but a more robust method to identify potential deadlocks would help a lot. We can do so through the use of progress graphs. These are informal sketches, where we can map all the mutex interactions, and can then deduce any potentially deadlocking behaviour. For an example looking to figure 10.1. In this type of graph we map each threads progression along the graph axis'. We only need to map any locking and unlocking operations, as anything else may be time consuming but is ultimately a non-blocking operation that will complete in a finite amount of time.

As each thread is sequential, its progress can be mapped by following along its axis, with any point in the graph being an expression of the combined states of both graphs. For example, the bottom left corner would be neither thread having started yet, with the top right corner being both threads having completed. The shaded blue section shows forbidden zones, where due to our mutexes is an impossible state to reach. There are two overlapping zones here, one for mutex A and one for mutex B, with them each overlapping in the middle of the graph. We can derive the locations of these zones by taking the coordinate of where each thread locks the mutex as the bottom left corner, and the coordinate of where each thread unlocks the mutex as the top right corner.

Two traces have been shown through the graph, one in green and one in red. The green shows a valid route through that does not enter any forbidden

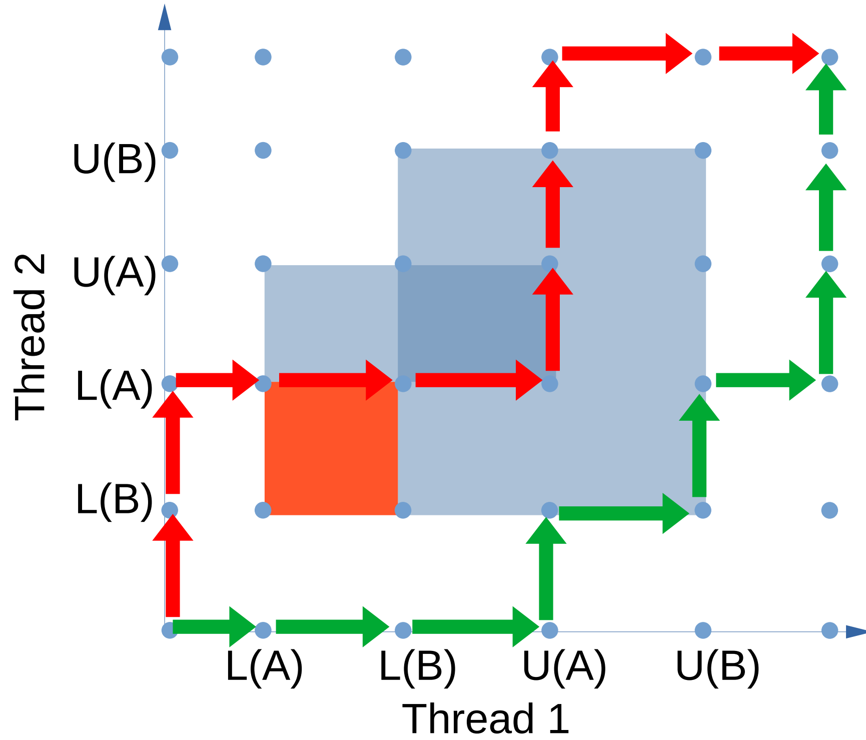


Figure 10.1: A progress graph of the two potentially deadlocking threads.

zone and so will produce a valid result. The red is an impossible trace as it enters a state within the forbidden zones, and so cannot occur in practice.

The concerning part of this graph is the red shaded area towards the bottom left of the graph. This shows a potential deadlock. As each thread can only progress linearly, our route through the graph can only be parallel to either axis. If our state ever enters the red zone then there is no way to escape as progress is blocked by the two forbidden zones. This makes it trivial to say that if we can draw a progress graph, if it none of these progress traps exist then our system is deadlock free. A solution to this is to reorder the operations our threads perform. This can be seen in figure 10.2 where both threads are now locking and unlocking the mutexes in the same order. No progress traps exist and any valid state has at least one path out of, therefore we are always deadlock free.

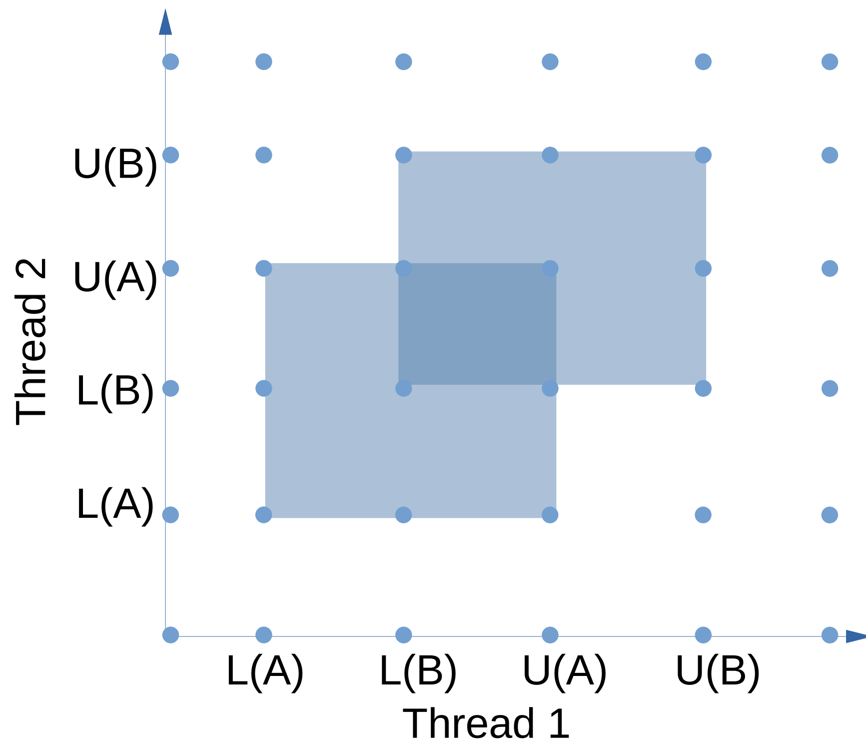


Figure 10.2: A progress graph of the two never deadlocking threads.

Chapter 11

Parallel Speedup and Scalability

While *concurrent programming* is often done to model the problem domain nicely (e.g. have a thread per connection to a web server), *parallel programming* is primarily concerned with speeding up our programs. This chapter will introduce nomenclature for talking about and comparing the performance of programs, and also discuss ways in which we can predict the potential performance advantage from parallelising a program.

11.1 Speedup

Suppose we are given some program and asked to speed it up. We then hack on it for a bit based on our knowledge of low-level programming. But how do we quantify our improvements? The standard approach to comparing the performance of two programs is by computing the *speedup* of one over the other.

11.1.1 Speedup in latency

The easiest way to quantify the performance of a program by itself is to run it and measure how long it takes. This is called program *latency* (often called *runtime*): how long from it starts until the result is ready? This is usually measured in *wall time*, because it corresponds to the real-world time we can measure with a clock on our wall. In contrast to this is *CPU time*, which is the total amount of time spent executing code on the CPUs we have available. When we parallelise a program, we decrease the wall time, but typically not the CPU time—16 CPUs that simultaneously run for 60 seconds equates 960 seconds of total CPU time, but will only have taken 60 seconds of wall time.

We usually have to put in effort to make sure that our time measurement is reliable. For example, we must make sure that we are measuring what we intend to measure—sometimes we do not wish to measure e.g. startup overhead, or loading data from files. It's also an easy mistake to make to measure CPU time rather than wall time, which will hide the advantage of parallelisation. Also, particularly with short-running programs, we must perform mul-

tiple measurements to average out random timing effects caused by random scheduling decisions taken by the operating system, or background tasks waking up and causing cache evictions. As a final concern, we must also make sure that we are compiling with optimisations, for example by passing `-O3` to the C compiler. You may be used to passing `-g`, which is good for debugging, but hinders the performance of the generated code.

Once we have reliable runtime measurements for both the original program and our modified program, we compare them by computing the *speedup*:

Definition 11.1 (Speedup in latency) *If T_1, T_2 are the runtimes of two programs P_1, P_2, then the* speedup in latency *of P_2 over P_1 is*

$$\frac{T_1}{T_2}$$

For example, if we have a sequential program that runs in $25s$ and we manage to write a parallel program that runs in $10s$ on our machine, then we compute the speedup of the parallel program as

$$\frac{25}{10} = 2.5$$

We would then say that the speedup we obtain is 2.5. Speedup is a dimensionless quantity, but it's common to write it with a tailing $\times$, as in $2.5\times$. The speedup formula can explain why programmers sometimes say "program A is twice as fast as B", when they really mean "program A runs in half the time as B"—they are talking about the speedup being 2.

11.1.2 Speedup in throughput

Latency speedup is useful for programs where the workload is *fixed*. But sometimes we are in a situation where the workload is infinite, for example in a long-running server that constantly processes new requests. Here latency is only meaningful within a single request, and to quantify the performance of the entire system, it is more interesting to look at the *throughput* of how many requests per time unit can be processed. Measuring throughput also allows us to compare the performance of programs that operate on different data sets.

The throughput Q is computed simply as the *workload* W processed in some time-span T:

$$Q = \frac{W}{T}$$

How we measure the workload depends on the concrete program. For a web server, we would measure requests. For matrix multiplication, we might measure total number of input elements accessed. Once we have computed throughput, we can then compute the speedup.

Definition 11.2 (Speedup in throughput) *If Q_1, Q_2 are the throughputs of two programs P_1, P_2, then the* speedup in throughput *of P_2 over P_1 is*

$$\frac{Q_2}{Q_1}$$

For example, suppose we have a program P_1 that can sum a megabyte in $69\mu s$, and a program P_2 that can sum a gigabyte in $28,589\mu s$. Since the workloads are different, we cannot directly compare their latency, but we can compute the throughputs as follows:

$$Q_1 = \frac{2^{20}B}{69\mu s} = 15196\text{B}/\mu\text{s} = 14.2\text{GiB/s}$$
$$Q_2 = \frac{2^{30}}{28589} = 37558\text{B}/\mu\text{s} = 35.0\text{GiB/s}$$

The speedup in throughput of P_2 over P_1 is

$$\frac{35.0\text{GiB/s}}{14.2\text{GiB/s}} = 2.46$$

Note that while lower numbers are better for latency, higher numbers are better for throughput. In both cases, a higher speedup is better.

11.2 Scalability

By *scalability* we mean how the system improves in its capacity (runtime or throughput) as we add more resources, such as more processors. It can also be used to describe how the performance changes as the problem size increases—this is essentially what big-O notation is for. With respect to parallelisation, we are interested in how the performance of a system changes as we add or exploit more processors. We distinguish two forms of scalability.

Definition 11.3 (Strong scaling) *How the runtime varies with the number of processors for a fixed problem size.*

Definition 11.4 (Weak scaling) *How the runtime varies with the number of processors for a fixed problem size relative to the number of processors.*

11.2.1 Amdahl's Law

Before we start on the often significant task of parallelising a program, or using a larger and more parallel computer to run it, it is worthwhile to estimate the potential performance gain. Unfortunately, it is not all parts of a program that benefit from increased parallelisation. For example, suppose a program needs 20 hours to run, but a 1-hour part of the program cannot possibly be

parallelised. This is not unlikely: perhaps that hour is spent reading configuration data, loading code, formatting human-readable reports, or waiting for the human operator to interact with the system somehow. Even if we optimise the program such that the optimisable 95% of the program runs in *zero time*, we have only achieved a speedup of 20.

Gene Amdahl inspired the now-famous *Amdahl's Law* [1] to describe the theoretical speedup from parallelisation:

Definition 11.5 (Amdahl's Law) *If p is the proportion of execution time that benefits from parallelisation, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by*

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

We can see that

$$S(N) \leq \frac{1}{1-p}$$

This means that the potential speedup by optimising part of a system is bounded by how dominant this part is in the overall runtime. It tells us that we should spend our time optimising the parts that take the most time to run. As fig. 11.1 shows, it is a rather pessimistic law—even in the case where 99% of the program can be parallelised, execution on 300 processors will give us a speedup of about 75 over a single processor.

While Amdahl's Law is usually applied to parallelisation, it can be used to characterise *any* situation where we are optimising a part of some system.

11.2.2 Gustafson's Law

As parallel supercomputers became more common in the 80s, researchers found that they routinely achieved speedup far in excess of what Amdahl's Law would predict. This is because Amdahl's Law is quite pessimistic, as it assumes that the workload stays *fixed* as we gain access to more computational resources. In practice, as workloads increase in size, the parallelisable fraction tends to *increase* in its share of the overall runtime. Also, when we get access to a larger machine, we tend not to be interested in solving our old problems faster, but in solving bigger problems in the same time as it took to solve our old problems. *Time* is the constant, not the workload.

Suppose we scale the runtime to be 1 and use s, p to indicate the fraction of this unit runtime spent in sequential and parallel code respectively on a parallel system with N threads. Then a sequential processor would require $s + N \times p$ time to execute the program. The scaled speedup of parallel execution is then

$$\frac{s + p \times N}{s + p} = s + p \times N = N + (1 - N) \times s$$

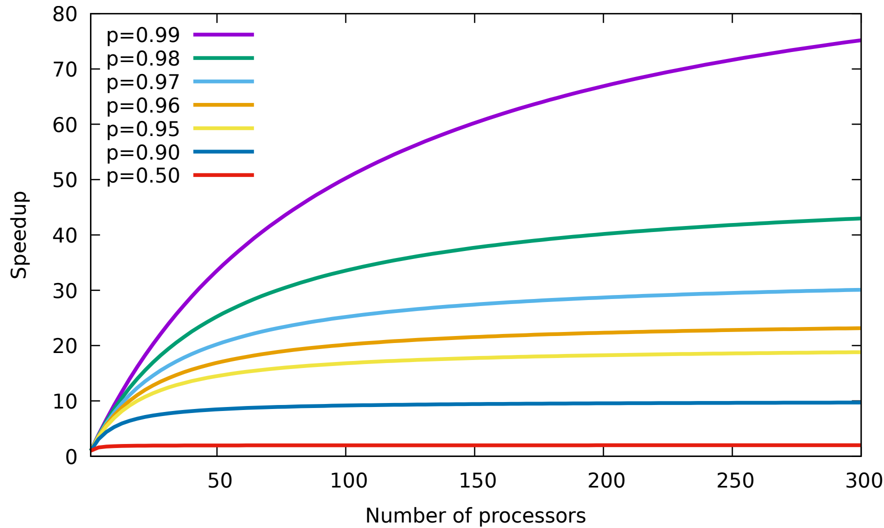


Figure 11.1: A graph of Amdahl's Law, plotted for various values of p.

This observation was first published by John L. Gustafson [3] and is therefore called Gustafson's Law:

Definition 11.6 (Gustafson's Law) *If s is the proportion of execution time that must be sequential, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by*

$$S(N) = N + (1 - N) \times s$$

Compared to Amdahl, Gustafson is much more of an optimist—as shown on fig. 11.2, Gustafson's Law plots as a *line*, meaning that the speedup as we add more processors is *linear*.

Neither Amdahl's nor Gustafson's Laws are *laws* in the common sense of the word. Despite providing conflicting predictions, they can both be true under different circumstances. Amdahl's Law tells us about the limitations of parallelism under a fixed workload, while Gustafson's Law tells us about the limitations of parallelism where we assume we the workload grows proportionally with the amount of parallelism. Broadly, Amdahl's Law predicts strong scalability, and Gustafson's law predicts weak scalability.

Both laws make significant simplifying assumptions—in practice, little scientific code consists of enormous fully parallel loops with completely independent iterations, but will tend to require some form of routine communication, proportional to the number of processors involved. Specifically, these laws tend to discount the nonlinear scaling of accessing large amounts data due to locality effects.

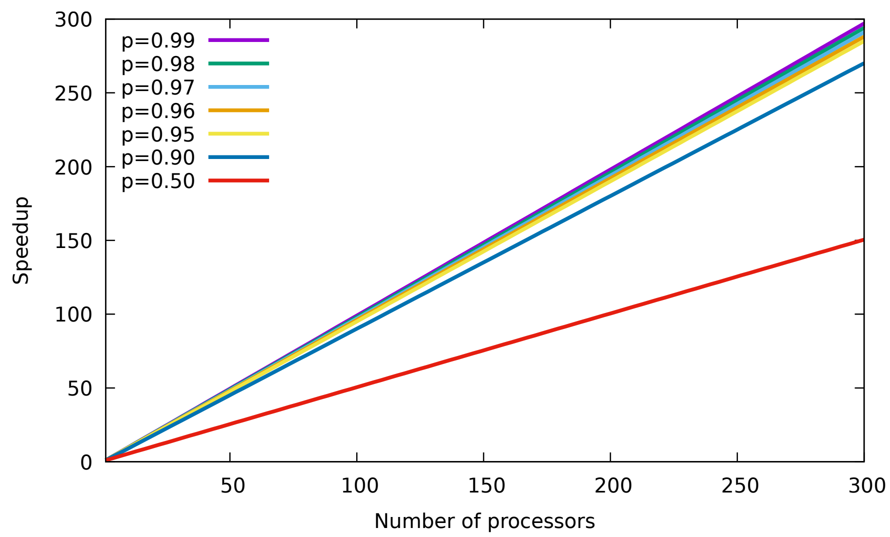


Figure 11.2: A graph of Gustafson's Law, plotted for various values of p (note that $p = 1 - s$).

Chapter 12

Loop Dependence Analysis

This chapter is adapted with permission from the PMPH Lecture Notes written by Cosmin Oancea.

So far, we have assumed that the user writes a fresh implementation of a known algorithm, and that it can be straightforwardly expressed as fully parallel loops, or as a simple reduction. However, there is a lot of legacy sequential scientific code written in imperative languages such as C++, Java, Fortran, and either the precise algorithm to which they correspond to (i) may have been forgotten (not documented), or (ii) a fresh implementation is infeasible (e.g., because it costs too much). At some point you may wish (or be asked) to parallelise such code to run efficiently on a certain hardware.

This will require you to:

1. Identify the loop nests[1] where most of the runtime is spent.
2. Parallelise these loops by reasoning at a low level of abstraction about which loops in the nest are parallel.
3. Decide on the manner in which loop nests can be re-written in order to optimise locality of reference, load balancing, thread divergence, etc.

The main source of inspiration for the material presented in this and chapter 13 has been the book "Optimizing compilers for modern architectures: a dependence-based approach" [4].

This chapter is organised as follows:

- Section 12.1 introduces various nomenclature, in particular the notion of cross-iteration dependency, and it shows how to summarise dependencies across the iteration space into a succinct representation, named direction vectors, that promotes reasoning about various code transformations.
- Section 12.2 presents a simple theorem that allows easy identification of loop-level parallelism.

[1] A *loop nest* is a collection of multiple loops nested within each other.

```
S1:   X  = ..
S2:   .. = X
```

(a) RAW: Read after write.

```
S1:   .. = X
S2:   X  = ..
```

(b) WAR: Write after read.

```
S1:   X = ...
S2:   X = ...
```

(c) WAW: Write after write.

Figure 12.1: Three different kinds of dependencies.

12.1 Direction vectors

We start by defining the various reasons why a program statement must be executed after some previous. C executes statements in *program order*—the order they occur in the source code. Dependence analysis is about distinguishing when the program order is crucial for correct execution, and in which cases it is arbitrary because C requires us to write statements in *some* order. If a statement S_2 *depends* on S_1, then S_2 must always be executed after S_1. Dependencies typically arise because the two statements interact with the same values, with at least one of the statements changing the value. When two statements are not dependent on each other, they can in principle be executed in parallel. Our goal is to use dependence analysis to identify when entire *loop iterations* are independent, which means that they can be executed in parallel.

The possible kinds of dependencies are depicted in fig. 12.1 between two statements S_1 and S_2, which reside for simplicity in the same *basic block*—a straight-line of code which is always entered by the first statement and is exited after the execution of the last statement. For example, the body of a loop can be considered a basic block if it contains no control flow and no `continue` textttbreak statements. The possible kinds of dependencies are as follows:

RAW (fig. 12.1a): refers to the case when a write to a register or memory location is followed, in program order, by a read from the same register or memory location; this is typically referred to as a read-after-write hazard in hardware-architecture nomenclature, and as a *true* dependency in loop-based analysis nomenclature. The word *true* refers to the fact that such a dependency denotes a producer-consumer relation, in which the value produced by S_1 is used in S_2. The producer-consumer relation is an algorithmic property of the program; such a dependency cannot be eliminated other than by changing the underlying algorithm.

WAR (fig. 12.1b): refers to the case when a read from a register or memory location is followed, in program order, by a write to the same register or memory location; this is referred to as a write-after-read hazard, and equivalently as an *anti* dependency. The problem here is that, if the two statements are reordered—meaning S_2 executes before S_1, then the value needed by S_1 is no longer available because it has been already overwritten by S_2.

WAW (fig. 12.1c): refers to the case when a write from a register or memory location is followed, in program order, by another write to the same reg-

ister or memory location; this is referred to as a write-after-write hazard, and equivalently as an *output* dependency. The problem here is that, if the two statements are reordered—meaning S_2 executes before S_1, then the final value stored in register or memory location is that of S_1 rather than that of S_2.

In what parallelism or loop analysis is concerned, we are primarily interested in analysing the (true, anti and output) *dependencies that occur across different iterations of the loop*. For example such a true dependency would correspond to the case in which an early iterations `i` writes/produces an array element that is subsequently read/consumed in a later iteration `j > i`.

In what parallelisation is concerned, the main limiting factor are the true dependencies—which correspond to an algorithmic property—because the anti and output dependencies can be typically eliminated by various techniques, as we shall see.

12.1.1 Loop notation and lexicographic ordering of iterations in a loop nest

In the following we are concerned with loop nests that consist of **`for`**-loops where the number of iterations (the *trip count*) can be determined when the loop is first entered. This is the case when:

1. The loop counter is not modified inside the loop.
2. The loop condition is of the form $i < n$, where i is the loop counter and n does not change during execution of the loop.
3. The loop counter is increased by a constant for each loop iteration.

Most **`for`**-loops you have written probably satisfy these conditions.

In the following we will also assume that iterations in a loop nest are represented by a vector, in which iterations numbers are written down from the corresponding outermost to the innermost loop in the nest, and are ordered *lexicographically*—i.e., are ordered consistently with the order in which they are executed in the (sequential) program. This means that in the loop nest below:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    ... loop-nest body ...
```

iteration $\vec{k}$=`(i=2,j=4)` is smaller than iteration $\vec{l}$=`(i=3,j=3)` (i.e., $\vec{k} < \vec{l}$), because the second iteration of the outer loop is executed before the third iteration of the outer loop, no matter what the iteration numbers are executed for the inner loop (of index `j`). In essence the iteration numbers of inner loops are only used to discriminate the order in the cases in which all the outer-loop iterations are equal, for example $\vec{k}$=`(i=3,j=3)` < $\vec{l}$=`(i=3,j=4)`

12.1.2 Dependency definition

The precise definition of a dependency between two statements located inside a loop nest is given below.

Definition 12.1 (Loop Dependency) *There is a dependency from statement S_1 to statement S_2 in a loop nest* if and only if *there exists loop-nest iterations $\vec{k}$, $\vec{l}$ such that $\vec{k} \leq \vec{l}$ and there exists an execution path from statement S_1 to statement S_2* such that:

1. *S_1 accesses some memory location M in iteration $\vec{k}$, and*

2. *S_2 accesses the same memory location M in iteration $\vec{l}$, and*

3. *one of these accesses is a write.*

In such a case, we say that S_1 is the source *of the dependence, and that S_2 is the* sink *of the dependence, because S_1 is supposed to execute before S_2 in the sequential program execution.*
Dependencies can be visually depicted by arrows pointing from the source to the sink of the dependence.

The definition basically says that in order for a dependency to exist, there must be two statements that access *the same memory location* and one of the accesses must be a write—two read instructions to the same memory location do not cause a dependency. The nomenclature denotes the statement that executes first in the program order as *the source* and the other as *the sink* of the dependency. We represent a dependency graphically with an arrow pointing from the source to the sink.

Optimisations for instruction-level parallelism (ILP)—meaning eliminating as much as possible the stalls from processor's pipeline execution[2]—typically rely on intra-iteration analyses (i.e., $\vec{k} = \vec{l}$). Higher-level optimisations, such as detection of loop parallelism, are mostly concerned with analysing inter-iteration dependencies (i.e., $\vec{k} \neq \vec{l}$). For example the main aim could be to disprove the existence of inter-iteration dependencies, such that different iterations may be scheduled out of order (in parallel) on different cores, while the body of an iteration is executed sequentially on the same core. In such a context, intra-iteration dependencies are trivially satisfied, and so are not very interesting.

12.1.3 Aggregating dependencies with direction vectors

Assume the three loops presented in fig. 12.2, which will be used as running example to demonstrate data dependence analysis and related transformations. We make the important observation that the code is not in three-address code

[2]Outside the scope of HPPS.

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1 : A[j][i] = A[j][i]...
```

(a)

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1 : A[j][i] = A[j-1][i-1]...
    S2 : B[j][i] = B[j-1][i]...
  }
```

(b)

```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1 : A[i][j] = A[i-1][j+1]...
```

(c)

Figure 12.2: Three simple running code examples that will be used to demonstrate data-dependency analysis and related transformation. Note that the statement labels S_i are not part of the code as such, but used to refer to specific statements in the text.

(TAC) form: a statement such as `A[j][i] = A[j][i] + 3` would correspond to three TAC or hardware instructions: one that loads from memory `tmp1 = A[j][i]`, followed by one that performs the arithmetic operation `tmp2 = tmp1 + 3`, followed by one that writes to memory `A[j][i] = tmp2`. Automated analysis is for simplicity usually carried out on programs in TAC form but, for brevity, our analysis will be carried out at the statement level. A human may start analysing dependencies:

- by depicting the iteration space in a rectangle in which the x axis and y axis correspond to iteration numbers of the inner loop j and outer loop i, respectively, and
- then by reasoning point-wise about what dependencies may happen between two iterations.

A graphical representation of the dependencies of the three running code examples is shown in fig. 12.3. They can be intuitively inferred as follows:

- For the loop in fig. 12.2a, different loop-nest iterations (i_1, j_1) and (i_2, j_2) necessarily read and write different array elements `A[`j_1`][`i_1`]` and `A[`j_2`][`i_2`]`. This is because our assumption is that $(i_1, j_1) \neq (i_2, j_2)$, hence it cannot be that both $i_1 = i_2$ and $j_1 = j_2$. As such, the representation of

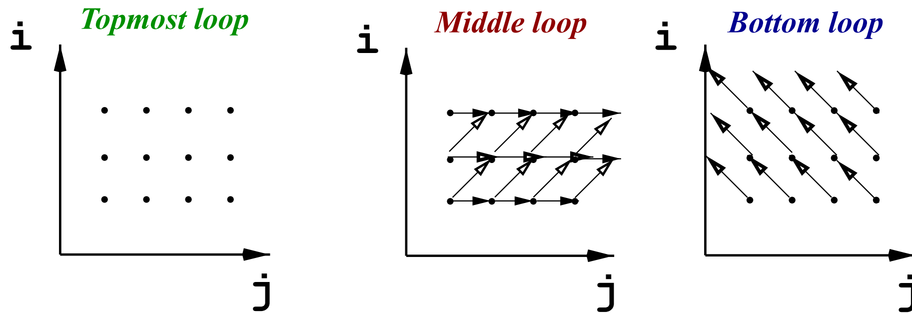


Figure 12.3: Graphical representation of the dependencies for the three running examples shown in fig. 12.2; the x and y axis correspond to the index of the inner and outer **do** loop, respectively.

dependencies should be a set of points (no arrows), meaning that all dependencies actually occur inside the same iteration—in fact they are anti intra-iteration dependencies (WAR) because `A[j][i]` is read first and then `A[j][i]` is written inside the same iteration.

- For the loop in fig. 12.2b we reason individually for statements S_1 and S_2 because each statement accesses (one) array `A` and `B`, respectively:

  S_1 : Let's take an iteration, say $(i_1 = 2, j_1 = 3)$, which *reads* element A[j_1-1][i_1-1] = `A[2][1]`. Since an iteration (i, j) always writes the element `A[i][j]`, we can reason that iteration $(i_2 = 1, j_2 = 2)$ will *write* the same element `A[2][1]`. It follows that we have discovered a true (RAW) dependency, depicted in the figure with and arrow, from the source iteration $(i_2 = 1, j_2 = 2)$—which writes `A[2][1]`—to the sink iteration $(i_1 = 2, j_1 = 3)$—which reads `A[2][1]`. This is because iteration $(1, 2) < (2, 3)$ according to the lexicographical ordering, and as such, the read happens after the write (RAW) in program order. One can individually reason for each point of the iteration space and fill it with oblique, forward-pointing arrows denoting true dependencies between different instances of statement S_1 (executing in different iterations).

  S_2 : Following a similar rationale, iteration $(i_1 = 2, j_1 = 3)$ *reads* element B[j_1-1][i_1] = `B[2][2]`, and iteration $(i_2 = 2, j_2 = 2)$ *writes* element `B[2][2]`. It follows that we have discovered a true (RAW) dependency with source $(i_2 = 2, j_2 = 2)$ and sink $(i_1 = 2, j_1 = 3)$, because $(2, 2) < (2, 3)$ in lexicographic ordering. Since $i_1 = i_2$ we depict the arrow parallel with the horizontal axis (that depicts values of j). One can fill in the rest of the iteration space with horizontal arrows.

- For the loop in fig. 12.2c we reason in a similar way: take iteration $(i_1 = 2, j_1 = 3)$ that *reads* element `A[2-1][3+1] = A[1][4]`. This

element is *written* by iteration ($i_2 = 1, j_2 = 4$). It follows that we have discovered a true (RAW) from source ($i_2 = 1, j_2 = 4$) to sink ($i_1 = 2, j_1 = 3$)—because the read happens in iteration $(2, 3)$ which comes after the write in iteration $(1, 4)$, i.e., $(1, 4) < (2, 3)$. Thus, one can fill in the iteration space with oblique, backward-pointing arrows, denoting true dependencies between instances of S_1 executing in different iterations.

We have applied above a human type of reasoning and, as a result, we have a graphical representation of all dependencies. However, such a reasoning is not suitable for automation because (i) the loop counts are statically unknown—they depend on the dataset—hence one cannot possibly represent an arbitrary large iteration space, and, more importantly, (ii) even if the loop counts would be statically known it is still inefficient to maintain and work with all this pointwise information.

A representation that promotes automated reasoning should succinctly capture the repeated pattern in the figure. Intuitively and imprecisely, for fig. 12.2a the pattern would correspond to a point, for fig. 12.2b it would correspond to two arrows—one oblique and one horizontal forward pointing arrows—and for fig. 12.2c it would correspond to an oblique, backward-pointing arrow. These patterns are formalized by introducing the notion of *direction vectors*.

Definition 12.2 (Dependency direction vector) *Assume there exists a dependency with source S_1 in iteration $\vec{k}$ to sink S_2 in iteration $\vec{l}$ ($\vec{k} \leq \vec{l}$). We denote by m the depth of the loop nest, we use i to range from $0, \ldots, m-1$, and we denote by x_i the ith element of some vector $\vec{x}$ of length m.*

The direction vector *between the instance of statement S_1 executed in some source iteration $\vec{k}$ and statement S_2 executed in sink iteration $\vec{l}$ is denoted by $\vec{D}(S_1 \in \vec{k}, S_2 \in \vec{l})$, and corresponds to a vector of length m, whose elements are defined as:*

$$D_i(S_1 \in \vec{k}, S_2 \in \vec{l}) = \begin{cases} < & \textit{if it is provably that } k_i < l_i, \\ = & \textit{if it is provably that } k_i = l_i, \\ > & \textit{if it is provably that } k_i > l_i, \\ \star & \textit{if } k_i \textit{ and } l_i \textit{ are statically uncomparable.} \end{cases} \tag{12.1}$$

The first three cases of the definition above assume that the ordering relation between k_i and l_i can be statically derived in a generic fashion (for any source k_i and l_i); if this is not possible than we use the notation $\star$ which *conservatively* assumes that any directions may be possible—i.e., star should be understood as simultaneous existence of all `<, =, >` directions. For example, the loop

```
for (int i = 0; i < N; i++)
  S1 : A[ X[i] ] = ...
```

would result in direction vector `[*]` corresponding to a potential output dependency (WAW), because the write access to `A[ X[i] ]` is statically unanalysable—for example under the assumption that the index array `X` is part of the dataset—and, as such, all direction vectors may possibly hold between various pairs of instances of statement S_1 executed in different iterations.

Note that the symbols `<, =, >` are *not* connected at all to the type of the dependency, e.g., true (RAW) or anti (WAR) dependency. The type of the dependency is solely determined by the operation of the source and that of the sink: If the source is a write statement and the sink is a read then we have a true (RAW) dependency; if the source is a read and the sink is a write then we have an anti (WAR) dependency; if both source and sink are writes then we have an output (WAW) dependency.

The meaning of the symbol > at some position i is that the source iteration at loop-level i is greater than the sink iteration at loop-level i. This case is possible, for example the code in fig. 12.2(c) shows a dependency with source iteration $(1, 4)$ and sink iteration $(2, 3)$. At the level of the second loop, we have $4 > 3$ hence the direction is > but still the source iteration is less than the sink iteration $(1, 4) < (2, 3)$ because of the first loop level. This observation leads to the following corollary:

Corollary 12.1 (Direction vector legality) *A direction vector is legal (well formed), if removing the = entries does* not *result in a leading > symbol, as this would mean that an iteration depends on a future iteration, and depending on a future event is considered impossible, and as such illegal.*

It remains to determine the sort of reasoning that can be applied to compute the direction vectors for the code examples in fig. 12.2:

The loop in fig. 12.2a: dependencies can occur only between instances of statement S_1, executed in different (or the same) iterations. We recall that, by the definition of dependency, the two (dependent) iterations must access the same element of `A` and at least one iteration should perform a write. Since statement S_1 performs a read and a write to elements of array `A`, two kinds of dependencies may occur:

WAW: an output dependency may be caused by two write accesses in two different iterations, denoted (i_1, j_1) and (i_2, j_2). The written element is thus $\texttt{A[j}_1\texttt{][i}_1\texttt{]}$, which must be the same as $\texttt{A[j}_2\texttt{][i}_2\texttt{]}$ for a dependency to exist. By eq. (12.1) this results in the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

which leads to direction vector `[=,=]`. Hence, an output dependency from S_1 to S_1 happens in the same iteration, but statement S_1 executes only one write access in the same iteration. The conclusion is that no output dependency can occur, hence the direction vector is discarded.

RAW: a true or anti dependency—we do not know yet which—will be caused by the read access from `A` and the write access to `A` in different (or same) iterations. Remember that a statement such as `A[j][i] = A[j][i] + 3` actually corresponds to three hardware instructions, hence either a cross- or an intra-iteration dependency will necessarily occur. Assume some iteration (i_1, j_1) reads from `A[`j_1`][`i_1`]` and iteration (i_2, j_2) writes to `A[`j_2`][`i_2`]`. In order for a dependency to exist, the memory location of the read and write must coincide; this results in the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

from which we can derive the direction vector: `[=,=]`. This implies that the dependency happens in the same iteration, hence it is an intra-iteration dependency. Furthermore, since the write follows the read in the instruction order of an iteration, this is an anti dependency (WAR).

For the loop in fig. 12.2b: dependencies may possibly occur between instances of statement S_1 and between instances of statement S_2. The case of output dependencies is disproved by a treatment similar to the bullet above. It remains to examine the dependency caused by a read and a write in different instances of S_1 and S_2, respectively:

S_1**:** assume iteration (i_1, j_1) and iteration (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this in eq. (12.1) results in the system

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

which necessarily means that $i_1 > i_2$ and $j_1 > j_2$. However, we do not know yet which iteration is the source and which is the sink. Assuming that (i_1, j_1) is the source results in the direction vector `[>,>]`, which is illegal by corollary 12.1, because a direction vector cannot start with the > symbol. It follows that our assumption was wrong: (i_2, j_2) is the source and (i_1, j_1) is the sink, which means that this is a cross-iteration true dependency (RAW)—because the sink iteration reads the element that was previously written by the source iteration—and its direction vector is `[<,<]`.

S_2**:** a similar rationale can be applied to determine that two instances of S_2 generate a true cross-iteration dependency (RAW), whose direction vector is `[=,<]`. In short, using the same notation results in the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

hence the source must be (i_2, j_2) and the sink must be (i_1, j_1) and the direction vector is `[=,<]`.

For the loop in fig. 12.2c: dependencies may possibly occur between instances of statement S_1. Assume iteration (i_1, j_1) and (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this into eq. (12.1) results in the system

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 + 1 = j_2 \end{cases}$$

which necessarily implies that $i_1 > i_2$ and $j_1 < j_2$. Choosing (i_1, j_1) as the source of the dependency results in direction vector `[>,<]`, which is illegal because it has > as the first non-= outermost symbol, as stated by corollary 12.1. It follows that (i_1, j_1) must be the sink and (i_2, j_2) must be the source, which results in the direction vector `[<,>]`, which is legal. Since the source writes and the sink reads, we have a true dependency (RAW). Moreover since the direction vector indicates that the source iteration is strictly less than the sink iteration, this is also a cross-iteration dependency.

Definition 12.3 (Dependency direction matrix) *A direction matrix is obtained by stacking together the direction vectors of all the intra- and cross-iteration dependencies of a loop nest (i.e., between any possible pair of write-write or read-read instruction instances).*

In conclusion, the direction matrices for the three running code examples:

Figure 12.2a: $\begin{cases} \texttt{[=,=]} \end{cases}$

Figure 12.2b: $\begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$

Figure 12.2c: $\begin{cases} \texttt{[<,>]} \end{cases}$

The following sections will show how the legality of powerful code transformations can be reasoned in a simple way in terms of direction vectors/matrices.

12.2 Determining loop parallelism

A loop is said to be parallel if its execution does not cause any (true, anti, or output) dependencies between iterations—the loop execution is assumed to be fixed in a specific iteration of an (potentially empty) enclosing loop context.

The following theorem states that a sufficient condition for a loop to be parallel is that for all the elements in the loop's corresponding direction-matrix column, it holds that the element is either = or there exists an outer loop whose corresponding direction is < (on that row). In the latter case we say that the

outer loop *carries* all the dependencies of the inner loop, i.e., fixing an iteration of the outer loop (think executing the outer loop sequentially) would guarantee the absence of cross-iteration dependencies in the inner loop.

Theorem 12.2 (Parallel loop) *We assume a loop nest denoted by $\vec{L}$, whose direction matrix is denoted by M and consists of m rows. A sufficient condition for a loop at depth k in $\vec{L}$, denoted L_k, to be parallel is that $\forall i \in \{0, \ldots m-1\}$ either $M[i][k]$ is = or there exists an outer loop at depth $q < k$ such that $M[i][q]$ is <. Proof left as an exercise.*

Theorem 12.2 claims to give only a sufficient condition for loop parallelism because it assumes that symbols such as $\star$ may be part of the direction vector elements—we recall that $\star$ conservatively assumes that all directions `<,=,>` may be possible. If $\star$ does not appear in the direction matrix, then the condition becomes *necessary* as well as sufficient. Let us analyse the parallelism of each loop in our running examples:

Figure 12.2a: The direction matrix is `[=,=]`, hence by theorem 12.2, both loops in the nest are parallel because all the directions are =.

Figure 12.2b: The direction matrix is

$$M = \begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$$

hence neither the outer nor the inner loop can be proven parallel by theorem 12.2. In the former case this is because $M[0, 0]$ is < and there is no other outer loop to carry dependencies. In the latter case this is because $M[1, 1]$ is < and the outer loop for that row has direction = (instead of <, which would have been necessary to carry the dependencies of the inner loop).

Figure 12.2c: The direction matrix is `[<,>]`, which means that the outer loop is not parallel—because it has a leading < direction—*but the inner loop is parallel* because the outer loop starts with < on the only row of the direction matrix, and, as such, it carries all the dependencies of the inner loop. To understand what this means, take a look again at the actual code in fig. 12.2c. Suppose we fix the outer iteration number to some value i. Then the read accesses always refer to row $i-1$ of matrix `A` and the write accesses always refer to row i of `A`; hence a cross-iteration dependency cannot happen in the inner loop because no matter the value of j, the read and write statement instances cannot possibly refer to the same location of `A`.

Chapter 13

Loop Transformations

This chapter is adapted with permission from the PMPH Lecture Notes written by Cosmin Oancea.

This chapter is organised as follows:

- Section 13.1 presents a simple theorem that gives necessary conditions for the safety of the transformation that interchanges two perfectly nested loops.
- Section 13.2 discusses the legality and the manner in which a loop can be distributed across the statements in its body.
- Section 13.3 discusses techniques for eliminating cross-iteration write-after-read and write-after-write dependencies.
- Section 13.4 introducing a simple transformation, named stripmining, which is always valid, and shows how block and register tiling can be derived as a combination of stripmining, loop interchange and loop distribution.

13.1 Loop interchange: legality and applications

Direction vectors are not used only for proving the parallel nature of loops, but can also enable powerful code restructuring techniques. For example they can be straightforwardly applied to determine whether it is safe to interchange two loops in a perfect loop nest[1]—which may result in better locality and even in changing an inner loop nature from dependent (sequential) to independent (parallel).

The following theorem gives a sufficient condition for the legality of loop interchange—i.e., for the transformation to result in code that is semantically equivalent to the original one.

[1] A perfect loop nest is a nest in which any two loops at consecutive depth levels are not separated by any other statements; for example all loop nests in fig. 12.2 are perfectly nested.

Theorem 13.1 (Legality of Loop Interchange) *A sufficient condition for the legality of interchanging two loops at depth levels k and l in a perfect nest is that interchanging columns k and l in the direction matrix of the loop nest* does not result *in a (leading) > direction as the leftmost non-= direction of any row.*

The theorem above shows that the legality of loop interchange can be determined solely by inspecting the result of permuting the direction matrix in the same way as the one desired for loops. For the rationale related to why a row-leading > direction is illegal, we refer the reader to corollary 12.1: a non-= leading > direction would correspond to depending on something that happens in the future: this currently seems impossible in our universe, and as such it signals an illegal transformation. The following corollary can be easily derived from theorem 13.1:

Corollary 13.2 (Interchanging a parallel loop inwards) *In a perfect loop nest, it is always safe to interchange a* parallel loop *inwards one step at a time (i.e., if the parallel loop is the kth loop in the nest then one can always interchange it with loop $k+1$, then with loop $k+2$, etc.).*

The corollary says that if we somehow know the parallel nature of a loop, then we can safely interchange it in the immediate inward position, without even having to build the dependence-direction matrix.

Let us analyse the legality of loop interchange for the three loop nests of our running example:

Figure 12.2a: The direction matrix is `[=,=]` and, as such, it is legal to interchange the two loops, because it would result in direction matrix `[=,=]`. Moreover applying loop interchange in this case is highly beneficial because it *optimises locality of reference*: the loop of index `i` appears in the innermost position after the interchange, which optimally exploits spatial locality for the write and read accesses to `A[j][i]`.

Figure 12.2b: The direction matrices are

$$M = \begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$$

and

$$M^{intchg} = \begin{cases} \texttt{[<,<]} \\ \texttt{[<,=]} \end{cases}$$

before and after interchange, respectively. It follows that the loop interchange is legal—because M^{intchg} satisfies theorem 13.1—and it also optimises spatial locality (as before). What is interesting about this example is that after the interchange, *the innermost loop has become parallel*, by theorem 12.2, because the outer loop caries all dependencies—the direction column corresponding to the outer loop consists only of < directions.

Figure 12.2c: The direction matrix is `[<,>]` and *interchanging the two loops is illegal* because the direction matrix obtained after the interchange `[>,<]` starts with a > direction; this would mean that the current iteration depends on a future iteration, which is impossible, hence the interchange is illegal.

13.2 Loop distribution: legality and applications

This section introduces a transformation, named loop distribution, where a loop is distributed across its statements. Potential benefits are:

- Loop distribution provides the bases for performing vectorisation: the innermost loop is distributed across its statements, and then the distributed loops are chunked (stripmined, section 13.4) by a factor that permits utilisation of processor's vector instructions.
- Loop distribution may enhance the degree of parallelism that can be statically mapped to the hardware. As discussed in section 10.3, OpenMP `collapse` clauses only apply to perfect loop nests. Distribution lets us split apart complex loop nests to create perfect nests of parallel constructs, which can then be parallelised efficiently with OpenMP.

Loop distribution requires the construction of a dependency graph, which is defined below.

Definition 13.1 (Dependency graph) *A dependency graph of a loop is a directed graph in which the nodes correspond to the statements of the loop nest and the edges correspond to dependencies. An edge is directed (points) from the source to the sink of the dependency, and is annotated with the direction corresponding to that dependence.*

In the case when the loop contains another inner loop, then the inner loop is represented as a single statement that conservatively summarises the behavior of all the statements of the inner loop.

The dependency graph of a loop can be used to characterise its parallel behavior:

Theorem 13.3 (Dependency cycle) *A loop is parallel* if and only if *its dependency graph does not have cycles.*

If the loop contains a cycle of dependencies, then it necessarily exhibits at least a cross iteration dependency (needed to form the cycle), and thus the loop is not parallel. The following theorem specifies how the transformation can be implemented:

Theorem 13.4 (Loop distribution) *Distributing a loop across its statements can be performed in the following way:*

1. *The dependency graph corresponding to the target loop is constructed.*

2. *The graph is decomposed into strongly-connected components (SCCs)*[2]*, and a new graph G' is formed in which the SCCs are nodes.*

3. *The loop can be safely distributed across its strongly-connected components, in the graph order of G'. Assuming a number k of SCCs, this means that the result of the transformation will be k loops, each containing the statements of the corresponding SCC. Inside an SCC, the statements remain in program order, but the distributed loops are ordered according to G'.*

4. *Array expansion (section 13.2.1) must be performed for the variables that*

   - *are either declared inside the loop or overwritten in each iteration (output dependencies),* **and**
   - *are used in at least two strongly-connected components.*

The theorem above says that the statements that are in a dependency cycle must remain in (form) one loop (which is sequential by theorem 13.3). As such, the loop can be distributed across groups of statements corresponding to the strongly connected components (SCC) of the dependency graph. If the graph has only one SCC then it cannot be distributed. The resulting distributed loops are written in the order dictated by the graph of SCCs. We demonstrate theorem 13.4 on the simple code example presented below:

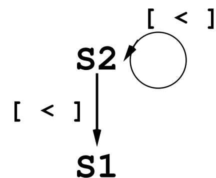
```
for (int i = 2; i < N; i++) {
  S1 : A[i] = B[i-2] ...
  S2 : B[i] = B[i-1] ...
}
```

The code has two dependencies:

$S_2 \to S_1$: In order for a dependency on `B` to exist the read from `B` in iteration i_1 of S_1 and the write to `B` in iteration i_2 of S_2 must refer to the same location. Hence $i_1 - 2 = i_2$, which means $i_1 > i_2$, hence S_2 is the source, S_1 is the sink and the direction vector is `[<]`;

$S2 \to S2$: similarly, there is a dependency between the read from `B` in S_2 and the write to `B` in S_2 of direction vector `[<]`.

The dependency graph is thus:



[2]A graph is said to be strongly connected if every vertex is reachable from every other vertex, i.e., a cycle. It is possible to find the strongly-connected components of an arbitrary directed graph in linear time $\Theta(V + E)$, where V is the number of vertices and E is the number of edges.

and it exhibits two strongly-connected components: one formed by statement S_2 and one formed by statement S_1. Loop distribution results in the following restructured code:

```
for (int i = 2; i < N; i++)
  S2 : B[i] = B[i-1] ...
for (int i = 2; i < N; i++)
  S1 : A[i] = B[i-2] ...
```

in which, according to the graph order, the loop corresponding to statement S_2 appears before the one corresponding to statement S_1. Note that this does not match the program order of statements S_1 and S_2 in the original program. Also note that the first loop is *not* parallel because the SCC consisting of S_2 has a (dependency) cycle, but the second loop is parallel because the SCC corresponding to S_1 does not have cycles.

If a loop is parallel then it can be straightforwardly distributed across its statements in program order because:

- by theorem 13.3, the loop dependency graph has no cycles and thereby each statement is a strongly connected component;
- the program order naturally respects all dependencies.

Corollary 13.5 (Parallel loop distribution) *A parallel loop can be directly distributed across each one of its statements. The resulting loops appear in the same order in which their corresponding statements appear in the original loop.*

13.2.1 Array expansion

Finally, it remains to demonstrate array expansion, mentioned in the fourth bullet of theorem 13.4. Assume the slightly modified code:

```
float tmp;
for (int i = 2; i < N; i++) {
  S1 : tmp = 2 * B[i-2];
  S2 : A[i] = tmp;
  S3 : B[i] = tmp + B[i-1];
}
```

Statements S_1 and S_3 are in a dependency cycle, because there is a dependency $S_3 \to S_1$ with direction $<$ caused by the write to and the read from array `B`, and a dependency $S_1 \to S_3$ with direction $=$ caused by `tmp`. Statement S_2 is not in a dependency cycle, but there is a dependency $S_1 \to S_2$, and hence its distributed loop should follow the distributed loop containing S_1 and S_3. If we do not perform array expansion, the distributed code:

```
float tmp;
for (int i = 2; i < N; i++) {
  S1 : tmp = 2 * B[i-2];
  S3 : B[i] = tmp + B[i-1];
}
for (int i = 2; i < N; i++) {
  S2 : A[i] = tmp;
}
```

does not respect the semantics of the original program because the second loop uses the same value of `tmp`—the one set by the last iteration of the first loop—while the original loop writes and then reads a different value of `tmp` for each iteration. We fix this by performing array expansion for `tmp`, which means that we must expand it with an array dimension equal to the loop count and replace its uses with corresponding indexing expressions of the expanded array. This results in the following *correct* code:

```
float tmp[N];
for (int i = 2; i < N; i++) {
  S1 : tmp[j] = 2 * B[i-2];
  S3 : B[i] = tmp + B[i-1];
}
for (int i = 2; i < N; j++) {
  S2 : A[i] = tmp[j];
}
```

Array expansion requires us to normalise the loop first—this means rewriting the loop such as its index starts from 0 and increases by 1 each iteration. This is why we have not written our example as **do** `i = 2, N+1`.

13.3 Eliminating false dependencies (WAR and WAW)

Anti and output dependencies are often referred to as *false* dependencies because they can be eliminated in most cases by copying or privatisation operations:

- Cross-iteration anti dependencies (WAR) typically correspond to a read from some original element of the array—whose value was set before the start of the loop execution—followed by an update to that element in a later iteration. As such, this dependency can be eliminated by copying (in parallel) the target array before the loop and rewriting the offending read access inside the loop such that it refers to the copy of the array.
- Cross-iteration output dependencies (WAW) can be eliminated by a technique named privatisation (or renaming), whenever it can be determined that *every read access* from a scalar or array location *is covered by an update* to that scalar or memory location that was previously performed *in the same iteration*. Semantically, privatisation moves the declaration

of the offending variable inside the loop, because it has been already determined that the read/used value was produced earlier in the same iteration.

- Reasoning based on direction vectors is limited to relatively simple loop nests; for example it is difficult to reason about privatisation by means of direction vectors.

13.3.1 Eliminating WAR dependencies by copying

Consider the simple C code below which rotates an array in the right dimension by one:

```
float tmp = A[0];
for (int i=0; i<N-1; i++) {
    A[i] = A[i+1]; // S1
}
A[N-1] = tmp;
```

The loop exhibits a cross-iteration anti dependency (WAR) $S_1 \rightarrow S_1$ (with direction vector `[<]`), and, as such, it is not safe to execute it in parallel. However, one can observe that the reads from `A` inside the loop correspond to the original elements of `A` before the loop, because they are rewritten in a later iteration. As such one can perform a copy of `A` before the loop, and replace the read access inside the loop to operate on the copy of array `A`. This preserves the original loop semantics and results in a parallel loop because the read and write accesses operate on different arrays, hence a dependency cannot occur. For example, using OpenMP:

```
float Acopy[N];
#pragma omp parallel for
for (int i=0; i<N; i++) {
    Acopy[i] = A[i];
}
tmp = A[0];
#pragma omp parallel for
for (int i=0; i<N-1; i++) {
    A[i] = Acopy[i+1];
}
A[N-1] = tmp;
```

Note that in a real program, we would allocate the `Acopy` array with `malloc()` rather than creating a potentially very large stack allocation.

13.3.2 Eliminating WAW dependencies by privatisation

Consider the contrived and ugly looking C code below:

```
int A[M];
for (int i=0; i<N; i++){
  for (int j=0, j<M; j++) { // writes slice A[0:M-1]
    A[j] = (4*i+4*j) % M;    // S1
  }
  for (int k=0; k<N; k++) { // reads A[j] where j∈{0,...M-1}
                            // because % denotes modulus op
    X[i][k] = X[i][k-1] * A[ A[(2*i+k)%M] % M];       // S2
  }
}
```

Analysing the cross-iteration dependencies of the outer loop, one can observe that there are frequent output dependencies $S_1 \to S_1$ of all directions $(\star)$, because, in essence, all elements of A at indices $0 \ldots M-1$ are (over)written in each iteration of the outer loop. This also causes frequent cross-iteration WAR and RAW dependencies between S_1 and S_2 of all directions $\star$ because S_2 reads some of the values of A which are written in S_1. The read access is also statically unanalysable because the index into A depends on a value of A (i.e., it is an indirect-array access `A[ A[...] ]`).

It would thus seem that this is a hopeless case and parallel execution is a pipe dream. Not so! Actually the rationale of how to transform the outer loop into a parallel one is quite simple. One may observe that each iteration of the outer loop writes the same indices of A, namely the ones belonging to the closed integral interval `[0,M-1]`. One may also observe that S_2 reads from A elements whose indices necessarily belong to `[0,M-1]`—due to the two modulus-M operations. As such, one may conclude that any value read in S_2 must have been produced in the same iteration of the outer loop (in the inner loop enclosing S_1).

It follows that it is safe to rewrite the loop in the following way:

1. Declare a new variable A′ of the same dimensions as A just inside the outer loop (or equivalently perform array expansion of array A′ with a new outer dimension of size N).

2. Replace all the uses of A in the outer loop by uses of A′. The resulting loop is safe to execute in parallel because there can be no dependencies on A′ since each iteration uses a different array A′.

3. As a last step, after the parallel execution of the loop terminates, one must copy (in parallel) the elements produced by the last iteration of the outer loop (i.e., `A′[0,...][M-1]` back to A.

The parallel OpenMP code that implements these steps is presented below:

```
int A[M];
#pragma omp for lastprivate(A)
for (int i=0; i<N; i++) {
    for(int j=0, j<M; j++) {
```

```
        A[j] = (4*i+4*j) % M;
    }
    for(int k=0; k<N; k++) {
        X[i][k]=X[i][k-1] * A[ A[(2*i+k) % M] % M];
    }
}
```

Declaring array `A` as private (by using the clause `private(A)`) would result in semantically performing steps (1) and (2) above. Using the `lastprivate(A)` clause instructs the OpenMP compiler to also perform step (3)—to copy back the privately-maintained result of `A` of the last executing iteration into the globally-declared array `A`.

Please also note that the OpenMP execution will not allocate a new `A'` for each iteration of the outer loop—this is actually equivalent to performing array expansion which is also applicable here—but instead it will *allocate a copy of* `A` *for each active thread*, thus significantly reducing the memory footprint and/or the number of (de)allocations.

Privatisation can be applied whenever one can prove that every read access in an iteration is covered by a previously-performed write access in the same iteration. Privatisation can be implemented by performing either array expansion or moving the declaration of the target variable from outside to inside the loop. However, it saves memory to allocate the private copy per active thread rather than per iteration, which is what OpenMP is doing.

13.4 Loop stripmining, block and register tiling

This section discusses several simple transformations that are going to be combined in various ways to optimise locality of reference (both temporal and spatial locality).

Stripmining refers to the following transformation, which is always safe to apply:

```
for(int i = 0; i<N; i++) {          for(int ii = 0; ii<N; ii+=T){
  iteration body               ⇒    for(int i=ii, i<min(ii+T,N); i++)
}                                          iteration body
                                    } }
```

In essence, a normalized loop is split into a perfect nest of two loops, in which the first loop goes with stride `T`, and the second one goes with stride `1`. Please notice that the resulting loop nest executes the same number of statements and in the same order as the original loop.

Block tiling refers to the transformation that stripmines several consecutive innermost loops in a perfect loop nest—named $l_{k+1} \ldots l_{k+n}$—and then interchanges inwards the resulting loops of stride 1. The transformation is valid/safe if in the original program it is safe to interchange any of the loops $l_{k+i},\ i \in \{1, \ldots, n-1\}$ in the innermost position. For example, the code below demonstrates block tiling a perfect loop nest of depth two:

```
for(i = 0; i<N; i++) {          for(ii=0; ii<N; ii+=T1) {
  for(j = 0; j<M; j++) {          for(jj=0; jj<M; jj+=T2) {
    iteration body       ⇒        for(i=ii; i<min(ii+T1,N); i++) {
  }                                   for(j=jj; j<min(jj+T2,M); j++) {
}                                       iteration body
                                } } } }
```

Unroll and jam refers to the transformation that partially unrolls one or more of the outer loops in a perfect nest and then fuses ("jams") the resulting loops. Equivalently, one can stripmine an outer loop, then interchange (distribute) it in the innermost position, then completely unroll it. The transformation is aimed at decreasing the number of memory loads and stores by storing to and reusing values from registers, and thus it is applied when the original loop nest contains data references that allow for temporal reuse—e.g., their indexes are invariant to some of the loops in the nest. Due to this, it is also known as "*register tiling*". We demonstrate the transformation on the matrix-matrix multiplication code below:

```
for(i=0; i<N; i++) {
  for(j=0; j<M; j++) {
    float c;
    c = 0.0;
    for(k=0; k<N; k++) {
      c += A[i][k] * B[k][j];
    }
    C[i][j] = c;
  }
}
```

The plan is to stripmine the loop of index `j` by a tile of size 2, and to interchange it to the innermost position, while performing the necessary loop distribution and array expansion:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
    float cs[2];
    for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] = 0.0;
    }
    for(k=0; k<N; k++) {
      for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] += A[i][k] * B[k][j];
    } }
    for(j=jj; j<min(jj+2,M); j++) {
        C[i][j] = cs[j-jj];
    }
} }
```

One can observe that the access `A[i][k]` is invariant to its immediately contained loop of index `j` and thus it can be hoisted outside it and saved into a register. Then the loops of index `j` can be unrolled, and array `cs` can be scalarized as well:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
    float c1, c2;
    if (jj   < M) c1 = 0.0;
    if (jj+1 < M) c2 = 0.0;
    for(k=0; k<N; k++) {
      float a;
      a = A[i,k];
      if (jj   < M) c1 += a * B[k][jj  ];
      if (jj+1 < M) c2 += a * B[k][jj+1];
    }
    if (jj   < M) C[i][jj  ] = c1;
    if (jj+1 < M) C[i][jj+1] = c2;
} }
```

In the resulted code, the accesses to the elements of `A` have been halved. We can similarly apply unroll and jam for the loop of index `i` with a tile size equal to `3`. This will cut down the accesses to `B` by a factor of 3. The resulted code is shown in fig. 13.1.

```
for(ii=0; ii<N; ii+=3) {
  for(jj=0; jj<M; jj+=2) {

    float c11, c12, c21, c22, c31, c32;

    if (ii   < N && jj   < M) c11 = 0.0;
    if (ii+1 < N && jj   < M) c21 = 0.0;
    if (ii+2 < N && jj   < M) c31 = 0.0;
    if (ii   < N && jj+1 < M) c12 = 0.0;
    if (ii+1 < N && jj+1 < M) c22 = 0.0;
    if (ii+2 < N && jj+1 < M) c32 = 0.0;

    for(k=0; k<N; k++) {

      float a1, a2, a3, b1, b2;

      if (ii   < N) a1 = A[ii  ][k];
      if (ii+1 < N) a2 = A[ii+1][k];
      if (ii+2 < N) a3 = A[ii+2][k];
      if (jj   < M) b1 = B[k   ][jj];
      if (jj+1 < M) b2 = B[k   ][jj+1];

      if (ii   < N && jj   < M) c11 += a1 * b1;
      if (ii+1 < N && jj   < M) c21 += a2 * b1;
      if (ii+2 < N && jj   < M) c31 += a3 * b1;
      if (ii   < N && jj+1 < M) c12 += a1 * b2;
      if (ii+1 < N && jj+1 < M) c22 += a2 * b2;
      if (ii+2 < N && jj+1 < M) c32 += a3 * b2;
    }

    if (ii   < N && jj   < M) C[ii  ][jj  ] = c11;
    if (ii+1 < N && jj   < M) C[ii+1][jj  ] = c21;
    if (ii+2 < N && jj   < M) C[ii+2][jj  ] = c31;
    if (ii   < N && jj+1 < M) C[ii  ][jj+1] = c11;
    if (ii+1 < N && jj+1 < M) C[ii+1][jj+1] = c21;
    if (ii+2 < N && jj+1 < M) C[ii+2][jj+1] = c31;
  }
}
```

Figure 13.1: Result of unroll-and-jam applied to matrix-matrix multiplication, where the first and second outer loops were tiled with sizes 3 and 2, respectively. The number of accesses to `A` and `B` has been reduced by a factor of $2\times$ and $3\times$, respectively, at the expense of introducing some conditional statements.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[2] Yukio-Pegio Gunji, Yuta Nishiyama, and Andrew Adamatzky. Robust soldier crab ball gate. In *AIP Conference Proceedings*, volume 1389, pages 995–998. American Institute of Physics, 2011.

[3] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[4] Ken Kennedy and John R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001.

[5] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.