CSCI3100: Software Engineering

Profiling, and code optimisation

Tak-Kei Lam, LLM

April 4, 2025

Table of contents

More on Profiling

Credits: Huge thanks to The University of Sheffield for their excellent materials "Performance Profiling & Optimisation by The University of Sheffield", which this chapter is based on.

In this chapter, we will also use the C programming language to demonstrate certain concepts that are abstracted away in Python.

Why profiling?

- Performance profiling is the process of analysing and measuring the performance of a program or script, to understand where time is being spent during execution.
- Profiling is useful for any code that will be running for a substantial period of time. As the code grows in complexity, it becomes increasingly difficult to estimate where time is being spent during execution. Profiling allows developers to narrow down where the time is being spent, to identify whether this is of concern or not.
- Profiling is a relatively quick process which can either provide developers the peace of mind that the code is efficient, or highlight the performance bottleneck. There is limited benefit to optimising components that may only contribute a tiny proportion of the

overall runtime. Identifying bottlenecks allows optimisation to be precise and efficient, potentially leading to significant speedups enabling faster research. In extreme cases, addressing bottlenecks has enabled programs to run hundreds or thousands of times faster!

- Increasingly, particularly with relation to High Performance Computing (HPC), attention is being paid to the energy usage of software. Profiling the software will provide the confidence that the software is an efficient use of resources.

When to Profile

- Profiling is most relevant to working code, when the software development process has reached a stage that the code works and are considering deploying it.
- Any code that will run for more than a few minutes over its lifetime, and is not a quick one-off script, can benefit from profiling.
- Profiling should be a relatively quick and inexpensive process. If there are no significant bottlenecks in the code, we can quickly be confident that the code is reasonably optimised. If we do identify a concerning bottleneck, further work to optimise the code and reduce the bottleneck could see significant improvements to the performance of the code and hence productivity.

Types of Profiler

There are multiple approaches to profiling, most programming languages have one or more tools available covering these approaches. Whilst these tools differ, their core functionality can be grouped into several categories. Some are listed below:

- Manual profiling with our good old friend `print()` Similar to using print() for debugging, manually timing sections of code can provide a rudimentary form of profiling.

```python
import time

t_a = time.monotonic()
# A: Do something
t_b = time.monotonic()
# B: Do something else
t_c = time.monotonic()
# C: Do another thing
t_d = time.monotonic()

mainTimer_stop = time.monotonic()
```

```python
print(f"A: {t_b - t_a} seconds")
print(f"B: {t_c - t_b} seconds")
print(f"C: {t_d - t_c} seconds")
```

- Function-Level Profiling

  Software is typically comprised of a hierarchy of function calls, both functions written by the developer and those used from the language's standard library and third party packages.

  Function-level profiling analyses where time is being spent with respect to functions. Typically, function-level profiling will calculate the number of times each function is called, and the total time spent executing each function, inclusive and exclusive of child function calls.

  This allows functions that occupy a disproportionate amount of the total runtime to be quickly identified and investigated. `cProfile` is one example of function-level profiler.

- Line-Level Profiling

  Function-level profiling may not always be granular enough, perhaps the software is a single long script, or function-level profiling highlighted a particularly complex function.

  Line-level profiling provides greater granularity, analysing where time is being spent with respect to individual lines of code.

  This will identify individual lines of code that occupy a disproportionate amount of the total runtime.

  However, accurate line-level profiling may not be always available because the code may be optimised by the compiler — lines of code are merged together.

Selecting An Appropriate Test Case

- Some **overhead is induced by the profiler**

  The act of profiling the code, collecting additional timing metrics during execution, will cause the program to execute slower. The slowdown is dependent on many variables related to both the code and the granularity of metrics being collected.

- Profiling, especially high resolution profiling, is demanding on CPU, memory and storage

  The longer the code runs, the more code that is being executed, the more data that will be collected. A profile that runs for hours could produce gigabytes of output data!

Therefore, it is important to select an appropriate test-case that is both representative of a typical workload and small enough that it can be quickly iterated. Ideally, it should take no more than a few minutes to run the profiled test-case from start to finish, however there may be circumstances where something that short is not possible.

For example, there may be a model which normally simulates a year in hourly time-steps. It would be appropriate to begin by profiling simulation of a single day. If the model scales over time, such as due to population growth, it may be pertinent to profile a single day later into a simulation if the model can be resumed or configured. A larger population is likely to amplify any bottlenecks that scale with the population, making them easier to identify.

Function-level Profiling

Function-level profiling analyses where time is being spent with respect to functions. Typically, function-level profiling will calculate the number of times each function is called, and the total time spent executing each function, inclusive and exclusive of child function calls.

This allows functions that occupy a disproportionate amount of the total runtime to be quickly identified and investigated.

Function-level Profiling - cProfile

```python
#long_series_of_func_call.py
import time

def a_1():
    for i in range(3):
        b_1()
    time.sleep(1)
    b_2()

def b_1():
    c_1()
    c_2()

def b_2():
    time.sleep(1)

def c_1():
    time.sleep(0.5)

```

```python
def c_2():
    time.sleep(0.3)
    d_1()

def d_1():
    time.sleep(0.1)

# Entry Point
a_1()
```

All of the methods except for b_1() call time.sleep(), this is used to provide synthetic bottlenecks to create an interesting profile.

```
    a_1() calls b_1() x3 and b_2() x1
    b_1() calls c_1() x1 and c_2() x1
    c_2() calls d_1()
```

Let's try to use cProfile to profile `long_series_of_func_call.py`:

```
 python -m cProfile -o out.prof src/profiling/long_series_of_func_call.py
```

The profiling statistics can be visualized with `kcachegrind` (introduced in the last chapter). There are other similar tools such as `snakeviz`.

To install `snakeviz`:

```
pip install snakeviz
```

Visualise the profiling statistics with `snakeviz`:

```
python -m snakeviz out.prof
```

Optimisation

Profiling and optimisation are closely related concepts in the context of software development and performance engineering. Profiling is the process of analysing a program to understand its behaviour, particularly in terms of resource usage such as CPU, memory, and I/O. Optimisation, on the other hand, involves making changes to the code or system configuration to improve performance based on the insights gained from profiling.

Some major performance optimisation methods are listed below:

1. Algorithm Optimisation: This is almost always the best method to improve the performance for most software.
   - Complexity Analysis: Analyse the time and space complexity of algorithms and data structures. Replace inefficient algorithms with more efficient ones, e.g. using a hash table instead of a list for lookups.
   - Parallelism: Implement parallel algorithms to take advantage of multicore processors.
   - Heuristics: Implement non-exact algorithms that may not always work efficiently, but can work quite good under certain conditions most of the time.
   - Randomized Algorithm: Design the algorithm in such a way that it can achieve efficient computation with higher probability if certain elements in the algorithm are randomized with certain distribution, e.g. choosing the pivot in quicksort randomly and uniformly so that the pivot can be away from the maximum or minimum numbers
2. Bottlenecks Identification:
   - Profiling Tools: Use profiling tools to identify which parts of the code consume the most resources.
   - Hotspots: Focus on hotspots, which are sections of the code where the program spends most of its time.
3. Code Optimisation And Memory Management:
   - Inlining: Use "function inlining" to reduce the overhead of function calls.
   - Loop Unrolling: Unroll loops to decrease the overhead of loop control.
   - Using L1/L2/L3 Cache: Try to use L1/L2/L3 cache to reduce main memory access
   - Best practices when using recursion: Write the recursive function in another way to avoid memory and run time explosion.
   - Garbage Collection Tuning: Optimise garbage collection settings and strategies to reduce pause times and memory overhead.
   - Memory Allocation: Use efficient memory allocation techniques and avoid memory leaks.
4. I/O Optimisation:
   - Buffering: Use buffering to reduce the number of I/O operations.
   - Asynchronous I/O: Implement asynchronous I/O to avoid blocking operations.
5. Concurrency and Parallelism:
   - Thread Pooling: Use thread pools to manage the lifecycle of threads efficiently.
   - Lock-Free Data Structures: Use lock-free data structures to reduce contention and improve scalability.

6. Caching:

    - Data Caching: Cache frequently accessed data to reduce the need for expensive computations or I/O operations.
    - Result Caching: Cache the results of expensive function calls.

By systematically profiling and identifying performance bottlenecks, and then applying targeted optimisations, the efficiency and responsiveness of the software can be significantly improved.

Rules for Profiling And Code Optimisation

1. Do not optimise when it is unnecessary
2. Do not profile or optimise when the code has not been proven functionally correct
3. Do not optimise when there is no regression test
4. Do not further optimise when it is time to stop

Performance VS. Maintainability

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. - Donald Knuth

This classic quote among computer scientists emphasises the importance of considering both performance and maintainability when optimising code and prioritising your optimisations.

While advanced optimisations may boost performance, they often come at the cost of making the code harder to understand and maintain. Even if you are working alone on private code, your future self should be able to easily understand the implementation. Hence, when optimising, always weigh the potential impact on both performance and maintainability.

Profiling is a valuable tool for prioritising optimisations. Should effort be expended to optimise a component which occupies 1% of the runtime? Or would that time be better spent optimising the most expensive components?

This doesn't mean you should ignore performance when initially writing code. Choosing the right algorithms and data structures is good practice. However, there is no need to obsess over micro-optimising every tiny component of your code—focus on the bigger picture.

Understanding Computer Program Execution and Optimisation

In order to optimise code for performance, it is necessary to have an understanding of what a computer is doing to execute it.

A high-level understanding of how your code executes, such as how Python and the most common data-structures and algorithms are implemented, can help you identify suboptimal approaches when programming. If you have learned to write code informally out of necessity, to get something to work, it's not uncommon to have collected some "unpythonic" habits along the way that may harm your code's performance.

These are the first steps in code optimisation, and knowledge you can put into practice by making more informed choices as you write your code and after profiling it.

The remaining content is often abstract knowledge, that is transferable to the vast majority of programming languages. This is because the hardware architecture, data-structures and algorithms used are common to many languages, and they hold some of the greatest influence over performance bottlenecks.

How a Computer Runs a Program

A computer runs a program by executing machine code instructions stored in the storage or in the memory. The process involves several steps:

1. Loading: The program is loaded from storage into memory.
2. Fetching: The CPU fetches the next instruction from memory.
3. Decoding: The CPU decodes the fetched instruction to understand what action is required.
4. Executing: The CPU executes the instruction.
5. Storing: The result of the execution is stored back in memory if needed.

Function Call and the Memory Stack

When a computer runs a function, it involves a series of well-defined steps that manage the function call, execution, and return. This process heavily relies on the **stack**, a special region of memory that stores temporary data such as function parameters, return addresses, and local variables. *Each function call creates a new **stack frame***, which is a data structure that contains all the information needed to manage the function's execution.

Function Call Mechanism

Function Call

A function call is an instruction that transfers control to a specific block of code defined as a function. This involves:

1. Saving the Return Address: The address of the next instruction in the calling function is saved so that control can return to it after the called function completes.
2. Passing Parameters: The arguments for the function are passed, typically via the stack or CPU registers.
3. Transferring Control: The program counter (PC) is set to the address of the called function, transferring control to it.

Function Return

When a function completes, control returns to the calling function. This involves:

1. Restoring the Return Address: The saved return address is retrieved.
2. Cleaning Up the Stack: The stack frame for the called function is destroyed.
3. Transferring Control Back: The PC is set to the return address, resuming execution in the calling function.

Example of Stack Growth and Shrinkage

Example in C

```c
#include <stdio.h>

void foo(int x) {
    int a = 10;
    printf("In foo: x = %d, a = %d\n", x, a);
}

void bar() {
    int b = 20;
    foo(b);
    printf("In bar: b = %d\n", b);
}

int main() {
    bar();
    return 0;
}
```

In this example, when `foo()` is called from `bar()`, a new stack frame is created for `foo()`. When `foo()` returns, its stack frame is destroyed, and control returns to `bar()`.

Initial State

Before any function is called, the stack is empty (or contains only the initial state of the program).

```
high memory address
+-------------------+
| Initial State     |
+-------------------+
low memory address
```

Step 1: `main()` Calls `bar()`

When `main()` calls `bar()`, a stack frame for `bar()` is created. (The stack grows from high memory address to low memory address in x86 architecture)

```
high memory address
+-------------------+
| Initial State     |
+-------------------+-+
  | Return Address    | <- Return address to `main()`
  +-------------------+
  | Saved FP          | <- Saved frame pointer (FP) of `main()`
  +-------------------+
  | Local Variable b  | <- Local variable `b` in `bar()`
  +-------------------+
low memory address
```

Step 2: `bar()` Calls `foo(b)`

When `bar()` calls `foo(b)`, a stack frame for `foo()` is created on top of the stack frame for `bar()`.

```
+-------------------+
| Initial State    |
+-------------------+
| Return Address   | <- Return address to `main()`
+-------------------+
| Saved FP         | <- Saved frame pointer (FP) of `main()`
+-------------------+
| Local Variable b | <- Local variable `b` in `bar()`
+-------------------+-+
  | Function Param x | <- Parameter `x` for `foo()`
  +-------------------+
  | Return Address   | <- Return address to `bar()`
  +-------------------+
  | Saved FP         | <- Saved frame pointer (FP) of `bar()`
  +-------------------+
  | Local Variable a | <- Local variable `a` in `foo()`
  +-------------------+
```

Step 3: `foo()` Executes and Returns

When `foo()` completes, its stack frame is destroyed, and control returns to `bar()`.

```
+-------------------+
| Initial State    |
+-------------------+
| Return Address   | <- Return address to `main()`
+-------------------+
| Saved FP         | <- Saved frame pointer (FP) of `main()`
+-------------------+
| Local Variable b | <- Local variable `b` in `bar()`
+-------------------+
```

Step 4: `bar()` Executes and Returns

When `bar()` completes, its stack frame is destroyed, and control returns to `main()`.

```
+-------------------+
| Initial State    |
+-------------------+
```

By understanding how the stack grows and shrinks with function calls, we can better appreciate the importance of stack management in program execution. Each function call creates a new stack frame, which contains all the necessary information for the function's execution. When the function completes, its stack frame is destroyed, and control returns to the calling function. This process ensures that each function has its own isolated environment for execution, allowing for modular and reusable code.

Understanding how a computer runs a function and manages the stack frame is crucial for writing efficient and optimized code. The stack frame plays a vital role in managing function calls, local variables, and return addresses. By leveraging this knowledge, developers can optimize function calls, manage memory more effectively, and improve overall program performance.

However, speed and memory are always in a tradeoff. To achieve the highest level of optimisation for a balanced speed-memory ratio, maximum speed, or minimal memory usage, we must also understand the computer's memory system.

Memory System

The computer's memory system is organized in a hierarchy to balance speed and cost. At the top of this hierarchy are the CPU caches (L1, L2, and L3), followed by the main memory (RAM), and finally, the disk storage.

1. L1 Cache: This is the smallest and fastest cache, located closest to the CPU cores. It typically has a size of a few kilobytes and is divided into separate instruction and data caches.
2. L2 Cache: Larger than L1, the L2 cache is also faster but slightly slower than L1. It is usually unified, meaning it stores both instructions and data, and its size ranges from a few hundred kilobytes to a few megabytes.
3. L3 Cache: This is the largest and slowest of the CPU caches, shared among all CPU cores. Its size can range from a few megabytes to tens of megabytes.
4. Random Access Memory (RAM): This is the main memory used by the computer to store data that is actively being used or processed. It is slower than the CPU caches but much larger, typically ranging from a few gigabytes to several tens of gigabytes.
5. Disk Storage: This is the slowest form of memory, used for long-term storage of data. It includes hard drives (HDDs) and solid-state drives (SSDs), with capacities ranging from hundreds of gigabytes to several terabytes.

Moving Data from Main Memory to L3, L2 and L1

Data in main memory is moved into the CPU caches (L1, L2, and L3) based on the principle of locality and the specific cache management policies implemented by the CPU. Here are the general scenarios when data is moved into the caches:

1. Temporal Locality When a piece of data is accessed, it is likely to be accessed again in the near future. This principle is known as temporal locality. When the CPU accesses data from main memory, it will load that data into the cache so that subsequent accesses can be faster.
2. Spatial Locality When a piece of data is accessed, it is likely that nearby data will also be accessed soon. This principle is known as spatial locality. When the CPU loads data into the cache, it often loads a block of data that includes the requested data and its surrounding data. This block is known as a cache line.
3. Cache Miss When the CPU tries to access data that is not currently in the cache (a cache miss), it will fetch the data from the next level of the memory hierarchy. If the data is not in L1, it will check L2, then L3, and finally main memory. Once the data is fetched, it is placed into the appropriate cache level(s) to speed up future accesses.
4. Cache Replacement Policies Caches have limited size, so when new data needs to be loaded into a full cache, some existing data must be evicted. The decision of which data to evict is made based on cache replacement policies such as Least Recently Used (LRU), First-In-First-Out (FIFO), or Random Replacement. These policies help maintain the most relevant data in the cache.
5. Prefetching Some CPUs use prefetching techniques to predict which data will be needed soon and load it into the cache ahead of time. This can be based on access patterns or specific instructions that hint to the CPU about future data needs.

Example of Cache Differences

While Python and most programming languages do not provide direct control over CPU caches, we can illustrate the concept of cache locality and its impact on performance using a simple example. This example demonstrates how accessing data in a way that leverages cache locality can be faster.

```python
# Function to access array elements sequentially
def sequential_access(array):
    sum = 0
    num_iterations = 100000
    for i in range(123456, len(array)): # start from 123456, to achieve the same difficulty
        sum += array[i]
        num_iterations -=1
        if num_iterations == 0:
            break
    return sum

# Function to access array elements with a large stride
```

```python
def random_access(array):
    # Not random actually
    sum = 0
    stride = 128
    num_iterations = 100000
    for i in range(0, len(array), stride):
        sum += array[i]
        num_iterations -=1
        if num_iterations == 0:
            break
    return sum
```

```python
from profiling.cache_locality import sequential_access, random_access
import pytest


@pytest.fixture
def large_array() -> list[int]:
    return [i for i in range(0, 100000000)]


def test_sequential_access(large_array, benchmark):
    benchmark(sequential_access, large_array)
    assert True == 1


def test_random_access(large_array, benchmark):
    benchmark(random_access, large_array)
    assert True == 1


# -------------------------------- benchmark: 2 tests -------------------------------
# Name (time in ms)               Min                  Max                 Mean
# ----------------------------------------------------------------------------------
# test_sequential_access     13.2866 (1.0)      14.0266 (1.0)      13.4407 (1.0)
# test_random_access         17.5095 (1.32)     25.1679 (1.79)     19.6962 (1.47)
# ----------------------------------------------------------------------------------
```

- `sequential_access()` accesses array elements sequentially, which is more likely to benefit from cache locality. When one element is accessed, the nearby elements are also loaded into the cache, making subsequent accesses faster.
- `random_access()` This function accesses array elements with a large stride, which is less

likely to benefit from cache locality. Each access may result in a cache miss, requiring data to be fetched from slower memory.

By understanding and leveraging the memory hierarchy, including the different levels of cache, we can write more efficient code that minimizes memory access times and maximizes performance.

Code Optimisation — Inlining

Function inlining is an optimisation technique where the function call is replaced with the actual code of the function. This can reduce the overhead of a function call, such as the time taken to set up the stack frame and jump to the function's code. However, it can also increase the size of the code, which might lead to other performance issues.

In Python, function inlining is not a built-in feature of the language or its standard interpreter (CPython). Python emphasizes readability and simplicity, and its dynamic nature makes inlining more complex compared to statically-typed languages like C or C++. However, manual inlining functions can be done to replace the function call with the function's code.

Without Inlining

```python
def add(a, b):
    return a + b

def main():
    x = 10
    y = 20
    result = add(x, y)
    print(f"The result is {result}")

if __name__ == "__main__":
    main()
```

In this example, the `add` function is called within the `main` function.

With Manual Inlining

```python
def main():
    x = 10
    y = 20
    # Inlining the add function
    result = x + y
    print(f"The result is {result}")

if __name__ == "__main__":
    main()
```

In the inlined version, the `add` function call is replaced with its body (`x + y`). This eliminates the overhead of the function call.

Considerations

1. Readability: Inlining can make the code harder to read and maintain, especially if the function is complex or used in multiple places.
2. Code Size: Inlining can increase the size of the code, which might lead to cache misses and other performance issues.
3. Dynamic Features: Python's dynamic features, such as first-class functions and decorators, can make inlining more challenging.

Tools and Techniques

While Python does not support automatic inlining, some tools and techniques can help optimise Python code:

1. Just-In-Time (JIT) Compilation: Tools like PyPy — another Python interpreter, use JIT compilation to optimise Python code at runtime, including inlining where beneficial.
2. Cython: Cython — the default Python interpreter, is a superset of Python that allows compiling Python code to C, which can then be optimised, including function inlining. There are some other tools such as Nuitka that can translate Python code into optimized C++ code, aiming to improve performance and reduce runtime overhead. It is essentially a Python-to-C++ compiler. It supports full compatibility with Python, allowing seamless integration with existing Python codebases while providing potential speed improvements and standalone executables.
3. Manual Optimisation: Profiling the code to identify bottlenecks and manually optimising critical sections can sometimes include inlining functions.

Code Optimisation — Loop Unrolling

Loop unrolling is an optimisation technique that involves replicating the loop body multiple times to reduce the overhead of loop control and increase instruction-level parallelism. It is mostly beneficial if some `dependency chains` can be broken, making it possible to execute instructions in parallel.

Example in Python

Consider a simple loop that sums the elements of a list:

```python
# Original loop
def sum_array(array):
    sum = 0
    for i in range(len(array)):
        sum += array[i]
    return sum
```

With loop unrolling, this loop might be transformed into:

```python
# Unrolled loop
def sum_array_unrolled(array):
    sum = 0
    n = len(array)
    for i in range(0, n, 4):
        sum += array[i]
        if i + 1 < n:
            sum += array[i + 1]
        if i + 2 < n:
            sum += array[i + 2]
        if i + 3 < n:
            sum += array[i + 3]
    return sum

# Is this implementation faster than the original one?
# It may be, because it does not need to increment the
# loop counter for every iteration; however, three
# additional `if` blocks are required. Benchmarking/profiling
# must be done to check whether there is performance gain.
# There are some other factors, such as cache sizes.
```

Another implementation with loop unrolling that allows `out of order execution` where the summations can be done in parallel if the CPU supports that:

```python
# Unrolled loop
def sum_array_unrolled_2(array):
    sum1 = 0
    sum2 = 0
    sum3 = 0
    sum4 = 0
    n = len(array)
    for i in range(0, n-3, 4):
        sum1 += array[i]
        sum2 += array[i+1]
        sum3 += array[i+2]
        sum4 += array[i+3]

    sum = sum1 + sum2 + sum3 + sum4;

    # if n%4 != 0, handle the remaining elements
    for i in range(n-3, n):
      sum += array[i]
    return sum
```

Considerations

Relationship with L1/L2/L3 Cache

1. Cache Line Utilization: By accessing contiguous memory locations in a single iteration, loop unrolling can improve cache line utilization, reducing cache misses and improving data locality.

2. Cache Size and Working Set: Unrolling loops can increase the working set of data. If this working set fits within the cache size, performance can improve. However, if it exceeds the cache size, it may lead to cache thrashing and degrade performance.

3. Prefetching: Loop unrolling can help hardware to prefetch by providing a more predictable access pattern, reducing memory latency.

Relationship with Function Stack Frame

1. Increased Code Size: Unrolling a loop increases the size of the function's code, which can lead to larger stack frames if the function has many local variables or if the unrolled loop contains many instructions.

2. Register Pressure: Unrolling a loop can increase the number of variables that need to be kept in registers. If there are not enough registers available, the compiler may spill some variables to the stack, increasing the stack frame size and potentially reducing performance.
3. Function Inlining: If loop unrolling is used together with function inlining, the code size and stack frame size will both be increased. These optimisations may turn out to be harmful to the performance.

Trade-offs

1. Code Size vs. Performance: Unrolling increases code size, which can lead to instruction cache misses and increased memory usage. The benefits of reduced loop overhead must outweigh the costs of increased code size.
2. Diminishing Returns: Beyond a certain point, further unrolling may not yield significant performance gains and can even degrade performance due to increased register pressure and cache effects.
3. Compiler Optimisations: Modern compilers often perform loop unrolling automatically. Developers should profile their code to determine if manual unrolling provides additional benefits.

Code Optimisation — Struct Field Alignment

In some programming languages such as C, struct field alignment refers to how data fields within a struct are arranged in memory. Proper alignment can improve access speed and reduce memory usage.

Suppose memory is aligned at units of 4 bytes in the following example in C – memory addresses are in muliple-of-four, or in other words, every memory load 4 bytes. Further suppose the data types have the following alignment requirements:

- A char (one byte) will be 1-byte aligned.
- An int (four bytes) will be 4-byte aligned.

In the `Aligned` struct, padding bytes may be added between fields to align `int b` on a 4-byte boundary. The `Packed` struct minimizes padding by reordering fields. Padding is only inserted when a structure member is followed by a member with a larger alignment requirement or at the end of the structure. By changing the ordering of members in a structure, it is possible to change the amount of padding required to maintain alignment.

```c
// size: 12 bytes
struct Aligned {
    char a; // 1 byte
    // some padding to fill 3 bytes
    int b;  // 4 bytes
    char c; // 1 byte
    // some padding to fill 3 bytes
};

// size: 8 bytes
struct Packed {
    char a; // 1 byte
    char c; // 1 byte
    // some padding to fill 2 bytes
    int b;  // 4 byte
};

struct PackedVersion2 {
    int b;  // 4 byte
    char a; // 1 byte
    char c; // 1 byte
    // some padding to fill 2 bytes
};
```

The total size of the struct `Packed` and `PackedVersion2` is now 8 bytes, reducing the amount of padding and potentially improving performance.

Struct field alignment is a technique used in computer programming to arrange the fields of a struct in memory in a way that adheres to the alignment requirements of the underlying hardware. Proper alignment can improve access speed and efficiency, as misaligned data can cause performance penalties due to additional memory accesses.

Considerations

Relationship with L1/L2/L3 Cache

1. Cache Line Utilization: Properly aligned data can be accessed more efficiently, reducing the number of cache lines needed to store the data. This can lead to fewer cache misses and better cache line utilization.
2. Cache Size and Working Set: By reducing the amount of padding, the overall size of the struct can be minimized. This can help keep the working set of data within the cache size, improving cache performance.

3. Prefetching: Properly aligned data can help hardware prefetchers by providing a more predictable access pattern, reducing memory latency.

Relationship with Function Stack Frame

1. Reduced Stack Frame Size: Properly aligned structs can reduce the overall size of the stack frame by minimizing padding. This can lead to more efficient use of stack space and reduce the likelihood of stack overflow.
2. Register Pressure: Properly aligned data can reduce register pressure by minimizing the number of memory accesses needed to load and store data. This can lead to more efficient use of registers and better performance.

Trade-offs

1. Code Complexity vs. Performance: If automatic struct field alignment is unavailable, rearranging struct fields for better alignment manually can sometimes make the code more complex. This is because alignment requirements are platform-dependent. Multiple versions of the code have to be maintained for multiple platforms according to the end users' working environment.
2. Compiler Support: Modern compilers of some programming languages such as Go and Rust often perform struct field alignment automatically. Developers should be aware of the capabilities of their development environment and profile their code to determine if manual alignment provides additional benefits.
3. Platform-Specific Alignment Requirements: Different platforms may have different alignment requirements. Developers should ensure that their code adheres to the alignment requirements of the target platform.

Code Optimisation — Tail Recursion

Tail recursion is a specific form of recursion where the recursive call is the last operation in the function. This allows for optimisations by the compiler or interpreter, where the **current function's stack frame can be reused** for the next function call, effectively **transforming the recursion into iteration** and reducing the overhead associated with function calls.

Example in C

Consider a simple recursive function to calculate the factorial of a number:

```c
// Non-tail recursive factorial function
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

This function is not tail-recursive because the multiplication operation (`n * factorial(n - 1)`) occurs after the recursive call.

A tail-recursive version of the factorial function would look like this:

```c
// Tail-recursive factorial function
int factorial_helper(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return factorial_helper(n - 1, n * acc);
    }
}

int factorial(int n) {
    return factorial_helper(n, 1);
}
```

In this version, the recursive call to `factorial_helper` is the last operation in the function, making it tail-recursive. The accumulator `acc` carries the result of the computation.

Tail Recursion in Python

Python does not support tail call optimisation (TCO) natively. This means that even if a function is written in a tail-recursive manner, Python will still create a new stack frame for each recursive call, which can lead to a stack overflow for deep recursion.

Why doesn't Python support tail call optimisation? "Proper traceback".

Example in Python

```python
# Tail-recursive factorial function in Python
# The Python intrepreter will not optimise the code, though
def factorial_helper(n, acc):
    if n == 0:
        return acc
    else:
        return factorial_helper(n - 1, n * acc)

def factorial(n):
    return factorial_helper(n, 1)
```

How to Achieve Tail Recursion

Transforming a recursive function into a tail-recursive function involves ensuring that the recursive call is the last operation in the function. The key steps to achieve this transformation involves:

1. Identify the Recursive Call: Locate the recursive call in the original function.
2. Accumulate Results: Introduce additional parameters (often called accumulators) to carry the intermediate results of the computation. This helps in passing the necessary state forward without needing to perform operations after the recursive call.
3. Move Operations Before the Recursive Call: Ensure that any operations that need to be performed are done before making the recursive call. The recursive call should be the last operation in the function.
4. Create a Helper Function: Often, it is useful to create a helper function that includes the additional parameters for accumulation. The original function can then call this helper function with the initial values for the accumulators.
5. Base Case: Ensure that the base case returns the accumulated result directly, without further computation.

Considerations

Relationship with L1/L2/L3 Cache

1. Cache Line Utilization: Tail recursion can improve cache line utilization by reducing the number of stack frames created, thereby reducing the amount of memory accessed and improving data locality.

2. Cache Size and Working Set: By reusing the same stack frame for each recursive call, tail recursion can reduce the working set of data that needs to be kept in the cache. This can lead to fewer cache misses and better cache performance.
3. Prefetching: Tail recursion can help hardware prefetchers by providing a more predictable access pattern, reducing memory latency.

Relationship with Function Stack Frame

1. Reduced Function Stack Size: Tail recursion allows the compiler to optimise the function by reusing the same stack frame for each recursive call. This reduces the overall function stack size and can prevent stack overflow for deep recursion.
2. Register Pressure: Tail recursion can reduce register pressure by minimizing the number of variables that need to be kept in registers across multiple stack frames. This can lead to more efficient use of registers and better performance.
3. Function Inlining: Inlining a tail-recursive function can further reduce the overall function stack size and improve performance.

Trade-offs

1. Code Complexity vs. Performance: Writing tail-recursive functions can sometimes make the code more complex. The benefits of reduced stack frame size and improved cache performance must outweigh the costs of increased code complexity.
2. Compiler Support: Not all compilers or interpreters support tail call optimisation. Developers should be aware of the capabilities of their development environment and profile their code to determine if tail recursion provides additional benefits.
3. Language Limitations: As mentioned, Python does not support tail call optimisation natively. Developers working in languages without TCO support should consider alternative optimisation techniques, such as iterative solutions.

Code Optimisation — Garbage Collection Tuning

Memory allocation and **memory management** are not free, but are very expensive.

Garbage collection (GC) is a form of automatic memory management that reclaims memory occupied by objects that are no longer in use by the program. Tuning garbage collection can significantly impact the performance of applications, especially those with high memory usage or real-time requirements.

In Python, garbage collection is managed by the built-in `gc` module, which provides an interface to the garbage collector for tuning and debugging purposes. Python uses a combination of reference counting and generational garbage collection to manage memory. Tuning garbage collection can significantly impact the performance of applications, especially those with high memory usage or real-time requirements.

Reference Counting

Reference counting is a technique where each object has an associated counter that tracks the number of references to it. When the reference count drops to zero, the object is deallocated because it is no longer accessible.

Example of Reference Counting

```python
import sys

a = []
b = a
print(sys.getrefcount(a))  # Output: 3 (a, b, and the argument to getrefcount)
del b
print(sys.getrefcount(a))  # Output: 2 (a and the argument to getrefcount)
```

Cyclic Reference Problem

Reference counting alone cannot handle cyclic references, where two or more objects reference each other, forming a cycle. These objects will never have their reference count drop to zero, leading to memory leaks.

Example of Cyclic Reference

```python
class Node:
    def __init__(self):
        self.next = None

a = Node()
b = Node()
a.next = b
b.next = a

# Both a and b reference each other, creating a cycle
```

In this example, `a` and `b` reference each other, forming a cycle. Even if we delete `a` and `b`, the reference counts will not drop to zero, and the memory will not be reclaimed. Hence, `generational garbage collection` is employed.

Generational Garbage Collection

Python's garbage collector uses a generational approach, where objects are divided into three generations based on their age:

1. Generation 0: Newly created objects.
2. Generation 1: Objects that have survived one garbage collection cycle.
3. Generation 2: Objects that have survived multiple garbage collection cycles.

The generational approach is based on the observation that most objects die young. By focusing on collecting younger objects more frequently, the garbage collector can optimise performance. By periodically examining objects in different generations and collecting those that are no longer reachable, generational garbage collection helps address the problem of cyclical references.

Example of Generational GC Tuning

Consider a Python application that processes large amounts of data and creates many temporary objects. Without proper garbage collection tuning, the application might experience performance issues due to frequent garbage collection cycles.

```python
# the garbage collector module for functions to manipulate garbage collection
import gc
import time

def process_data(data):
    # Simulate data processing
    temp_list = [i * 2 for i in data]
    return sum(temp_list)

def main():
    # enables debugging output, which provides
    # statistics about garbage collection cycles.
    gc.set_debug(gc.DEBUG_STATS)

    # To configure how often GC should run:
    #
    # The first value (700) is the threshold for the number of allocations
    # minus the number of deallocations before the collector runs.
    #
    # The second and third values (10, 5) are thresholds for the 1st
    # and 2nd generations of the garbage collector:
    #   - Generation 1 is only run if generation 0 has been run 10 times
    #   - Generation 2 is only run if generation 1 has been run 5 times
    #
    # In summary:
    # Generation 0 runs every 700 (allocation-deallocation) cycles,
    # so generation 1 runs every 7000 cycles,
    # and generation 2 runs every 35000 cycles.
    #
    # The values have to be tuned for each working environment for performance
    gc.set_threshold(700, 10, 5)

    data = list(range(1000000))
    start_time = time.time()

    for _ in range(10):
        result = process_data(data)
        print(f"Result: {result}")

    end_time = time.time()
    print(f"Processing time: {end_time - start_time} seconds")

if __name__ == "__main__":
    main()
```

Tuning Garbage Collection in Python

Garbage collection tuning involves adjusting the thresholds and parameters to balance the trade-offs between memory usage and performance. Here are some key considerations:

1. Thresholds: Adjusting the thresholds can control how frequently the garbage collector runs. Higher thresholds can reduce the frequency of garbage collection cycles, potentially improving performance but increasing memory usage. Lower thresholds can increase the frequency of garbage collection cycles, potentially reducing memory usage but impacting performance.

2. Generations: Python's garbage collector uses a generational approach, where objects are divided into three generations based on their age. Tuning the thresholds for each generation can help optimise the performance of the garbage collector.

3. Monitoring and Profiling: Use the `gc` module's debugging features to monitor and profile the garbage collector's behaviour. This can help identify performance bottlenecks and guide tuning efforts.
4. Application-Specific Tuning: The optimal garbage collection settings can vary depending on the application's workload and memory usage patterns. Experiment with different settings and profile the application's performance to find the best configuration.

Considerations

Relationship with L1/L2/L3 Cache

1. Cache Line Utilisation: Efficient garbage collection can improve cache line utilisation by reducing the number of cache lines needed to store data. This can lead to fewer cache misses and better cache performance.
2. Cache Size and Working Set: By optimising garbage collection, the overall memory footprint of the application can be reduced. This helps keep the working set of data within the cache size, improving cache performance.
3. Prefetching: Efficient garbage collection can help hardware prefetchers by providing a more predictable access pattern, reducing memory latency.

Relationship with Function Stack Frame

1. Reduced Stack Frame Size: Efficient garbage collection can reduce the overall size of the stack frame by minimising the number of temporary objects (aka short-lived objects) and memory allocations. This can lead to more efficient use of stack space and reduce the likelihood of stack overflow.
2. Register Pressure: Efficient garbage collection can reduce register pressure by minimising the number of memory accesses needed to load and store data. This can lead to more efficient use of registers and better performance.
3. Function Inlining: Efficient garbage collection can interact with function inlining (another optimisation technique where function calls are replaced with the function's body). Inlining functions with efficient garbage collection can further reduce the stack frame size and improve performance.

Trade-offs

1. Application-Specific Tuning: The optimal garbage collection settings can vary depending on the application's workload and memory usage patterns. Developers should profile their code to determine if manual tuning provides additional benefits.
2. Monitoring and Profiling: Use the `gc` module's debugging features to monitor and profile the garbage collector's behaviour. This can help identify performance bottlenecks and guide tuning efforts.

Code Optimisation — Memory Pool for Reducing The Frequency of Memory Allocation

A memory pool is a block of memory that is reserved for dynamic allocation. Instead of allocating and deallocating memory frequently, which can be inefficient and lead to fragmentation, a memory pool allows for more efficient memory management by allocating a large block of memory at once and then managing smaller allocations within that block. Memory pools are particularly useful in real-time systems, embedded systems, and applications where performance is critical. They are particular useful in situations where objects are created and destroyed frequently and have short lifetimes.

An Example of Using Memory Pool in C with Custom Structs

Let's define a custom struct and use the memory pool to allocate and deallocate instances of this struct.

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

#define POOL_SIZE 1024  // Total size of the memory pool
#define BLOCK_SIZE 32   // Size of each block

typedef struct MemoryPool {
    unsigned char pool[POOL_SIZE];
    unsigned char *freeList[POOL_SIZE / BLOCK_SIZE];
    int freeCount;
} MemoryPool;

typedef struct CustomStruct {
    int id;
    char name[24];
```

```c
} CustomStruct;

void initMemoryPool(MemoryPool *mp) {
    mp->freeCount = POOL_SIZE / BLOCK_SIZE;
    for (int i = 0; i < mp->freeCount; i++) {
        mp->freeList[i] = mp->pool + i * BLOCK_SIZE;
    }
}

void *allocateBlock(MemoryPool *mp) {
    if (mp->freeCount == 0) {
        return NULL;  // No free blocks available
    }
    return mp->freeList[--mp->freeCount];
}

void deallocateBlock(MemoryPool *mp, void *block) {
    if (mp->freeCount < POOL_SIZE / BLOCK_SIZE) {
        mp->freeList[mp->freeCount++] = block;
    }
}

int main() {
    MemoryPool mp;
    initMemoryPool(&mp);

    CustomStruct *cs1 = (CustomStruct *)allocateBlock(&mp);
    CustomStruct *cs2 = (CustomStruct *)allocateBlock(&mp);

    if (cs1) {
        cs1->id = 1;
        strcpy(cs1->name, "Struct1");
        printf("Allocated cs1: id=%d, name=%s\n", cs1->id, cs1->name);
    }

    if (cs2) {
        cs2->id = 2;
        strcpy(cs2->name, "Struct2");
        printf("Allocated cs2: id=%d, name=%s\n", cs2->id, cs2->name);
    }

    deallocateBlock(&mp, cs1);
    deallocateBlock(&mp, cs2);

    return 0;
}
```

Considerations

Relation to L1/L2/L3 Cache and Function Stack Frame

Memory pools can significantly improve cache performance. Here's how:

1. Cache Locality: By allocating memory from a contiguous block, memory pools can improve spatial locality, making it more likely that accessed memory is already in the cache. This is beneficial for L1, L2, and L3 caches, which are designed to speed up access to frequently used data.
2. Reduced Fragmentation: Memory pools reduce fragmentation by reusing fixed-size blocks. This can help maintain a more predictable memory access pattern, which is beneficial for cache performance.
3. Stack Frame Efficiency: When functions allocate memory on the stack, they benefit from the stack's LIFO (Last In, First Out) nature, which is cache-friendly. However, for dynamic memory allocation, using a memory pool can provide similar benefits by ensuring that memory allocations are close together in memory, improving cache hits.

In summary, memory pools can enhance performance by improving cache locality and reducing fragmentation, making them a valuable technique in performance-critical applications.

Memory Pool in Python

Implementing a memory pool in Python can be challenging due to several reasons, primarily related to Python's high-level abstractions and memory management mechanisms. Let's delve into these challenges and the abstractions provided by Python.

Challenges in Implementing Memory Pool in Python

1. Garbage Collection:
   - Python uses automatic garbage collection to manage memory, primarily through reference counting and cyclic garbage collection. This means that Python automatically handles the allocation and deallocation of memory for objects, making it more difficult to manually manage memory than in lower-level languages like C or C++.

2. High-Level Abstractions:
   - Python abstracts away many of the low-level details of memory management. For example, Python objects are dynamically typed and can grow or shrink in size, making it hard to predict and manage memory usage manually.
3. Global Interpreter Lock (GIL):
   - The GIL in CPython ensures that only one thread executes Python bytecode at a time. This can complicate the implementation of a memory pool, especially in multi-threaded applications, as thread safety must be ensured.
4. Lack of Direct Memory Access:
   - Python does not provide direct access to memory addresses or pointers, which are often used in memory pool implementations in lower-level languages. This limits the ability to manually allocate and deallocate memory blocks.

Code Optimisation — Object Pool for Reducing The Frequency of Memory Allocation

The Object Pool pattern is a design pattern that is used to manage the reuse of objects that are expensive to create. Instead of creating and destroying objects on demand, an object pool maintains a collection of reusable objects. When a client needs an object, it retrieves one from the pool. When it is done with the object, it returns it to the pool for future reuse. An object pool is in fact very similar to a memory pool, except the unit is an object instead of a block of bytes.

Let's implement a simple object pool in Python. We'll create a pool for a custom class `Resource`.

```python
import queue

class Resource:
    def __init__(self, name):
        self.name = name

    def reset(self):
        # Reset the state of the resource if needed
        pass

class ObjectPool:
    def __init__(self, max_size):
        self._pool = queue.Queue(maxsize=max_size)
        for _ in range(max_size):
            self._pool.put(Resource(f"Resource-{_}"))

    def acquire(self):
        try:
            resource = self._pool.get_nowait()
        except queue.Empty:
            raise Exception("No resources available")
        return resource

    def release(self, resource):
        resource.reset()
        self._pool.put(resource)

# Example usage
if __name__ == "__main__":
    pool = ObjectPool(max_size=3)

    # Acquire a resource from the pool
    resource1 = pool.acquire()
    print(f"Acquired: {resource1.name}")

    # Use the resource
    # ...

    # Release the resource back to the pool
    pool.release(resource1)
    print(f"Released: {resource1.name}")

    # Acquire another resource
    resource2 = pool.acquire()
    print(f"Acquired: {resource2.name}")
```

Code Optimisation — Language Specific Techniques

Core idea: There are numerous approaches to executing an algorithm. Out of all the potential methods of implementation, it is frequently more efficient to leverage the principles, idioms, and built-in functions of the programming language being used.

In this example, we are employing Python to demonstrate how performance enhancements can be realised by developing the software in a style that is characteristic of Python, commonly known as writing in a *pythonic way*.

However, these concepts are universally applicable to all programming languages, provided we have a clear understanding of:

- The problem at hand
- The fundamental workings of a computer
- The specific features and properties of the programming language being utilised

Built-in Functions of Python

We might think to sum a list of numbers by using a for loop, as would be typical in C, as shown in the function manualSumC() and manualSumPy() below.

```python
import random
from timeit import timeit

N = 100000  # Number of elements in the list

# Ensure every list is the same
random.seed(12)
my_data = [random.random() for i in range(N)]


def manualSumC():
    n = 0
    for i in range(len(my_data)):
        n += my_data[i]
    return n

def manualSumPy():
    n = 0
    for evt_count in my_data:
        n += evt_count
    return n

def builtinSum():
    return sum(my_data)


repeats = 1000
print(f"manualSumC: {timeit(manualSumC, globals=globals(), number=repeats):.3f}ms")
print(f"manualSumPy: {timeit(manualSumPy, globals=globals(), number=repeats):.3f}ms")
print(f"builtinSum: {timeit(builtinSum, globals=globals(), number=repeats):.3f}ms")
```

Even just replacing the iteration over indices (which may be a habit you've picked up if you first learned to program in C) with a more pythonic iteration over the elements themselves speeds up the code by about 2x. But even better, by switching to the built-in sum() function our code becomes about 8x faster and much easier to read while doing the exact same operation!

```
manualSumC: 1.624ms
manualSumPy: 0.740ms
builtinSum: 0.218ms
```

This is because built-in functions (i.e. those that are available without importing packages) are typically implemented in the CPython back-end, so their performance benefits from bypassing the Python interpreter.

In particular, those which are passed an iterable (e.g. lists) are likely to provide the greatest benefits to performance. The Python documentation provides equivalent Python code for many of these cases.

- `all()`: boolean and of all items
- `any()`: boolean or of all items
- `max()`: Return the maximum item
- `min()`: Return the minimum item
- `sum()`: Return the sum of all items

Example: Searching An Element in A List

A simple example of this is performing a linear search on a list. In the following example, we create a list of 2500 integers in the (inclusive-exclusive) range [0, 5000). The goal is to search for all even numbers within that range.

The function manualSearch() manually iterates through the list (ls) and checks each individual item using Python code. On the other hand, operatorSearch() uses the in operator to perform each search, which allows CPython to implement the inner loop in its C back-end.

```python
import random
from timeit import timeit

N = 2500  # Number of elements in list
M = 2  # N*M == Range over which the elements span

def generateInputs():
    random.seed(12)  # Ensure every list is the same
    return [random.randint(0, int(N*M)) for i in range(N)]

```

```python
def manualSearch():
    ls = generateInputs()
    ct = 0
    for i in range(0, int(N*M), M):
        for j in range(0, len(ls)):
            if ls[j] == i:
                ct += 1
                break

def operatorSearch():
    ls = generateInputs()
    ct = 0
    for i in range(0, int(N*M), M):
        if i in ls:  # Pay attention to this line
            ct += 1

repeats = 1000
gen_time = timeit(generateInputs, number=repeats)
print(f"manualSearch: {timeit(manualSearch, number=repeats)-gen_time:.2f}ms")
print(f"operatorSearch: {timeit(operatorSearch, number=repeats)-gen_time:.2f}ms")
```

This results in the manual Python implementation being 5x slower, doing the exact same operation!

```
manualSearch: 152.15ms
operatorSearch: 28.43ms
```

Lists in Python

Lists are a fundamental data structure within Python.

It is implemented as a form of dynamic array found within many programming languages by different names (C++: std::vector, Java: ArrayList, R: vector, Julia: Vector).

They allow direct and sequential element access, with the convenience to append items.

This is achieved by internally storing items in a static array. This array however can be longer than the list, so the current length of the list is stored alongside the array. When an item is appended, the list checks whether it has enough spare space to add the item to the end. If it doesn't, it will re-allocate a larger array, copy across the elements, and deallocate the old array. The item to be appended is then copied to the end and the counter which tracks the list's length is incremented.

Typically, a list is implemented in the way that an additional 10+% storage space is pre-allocated, to prepare for future growth.

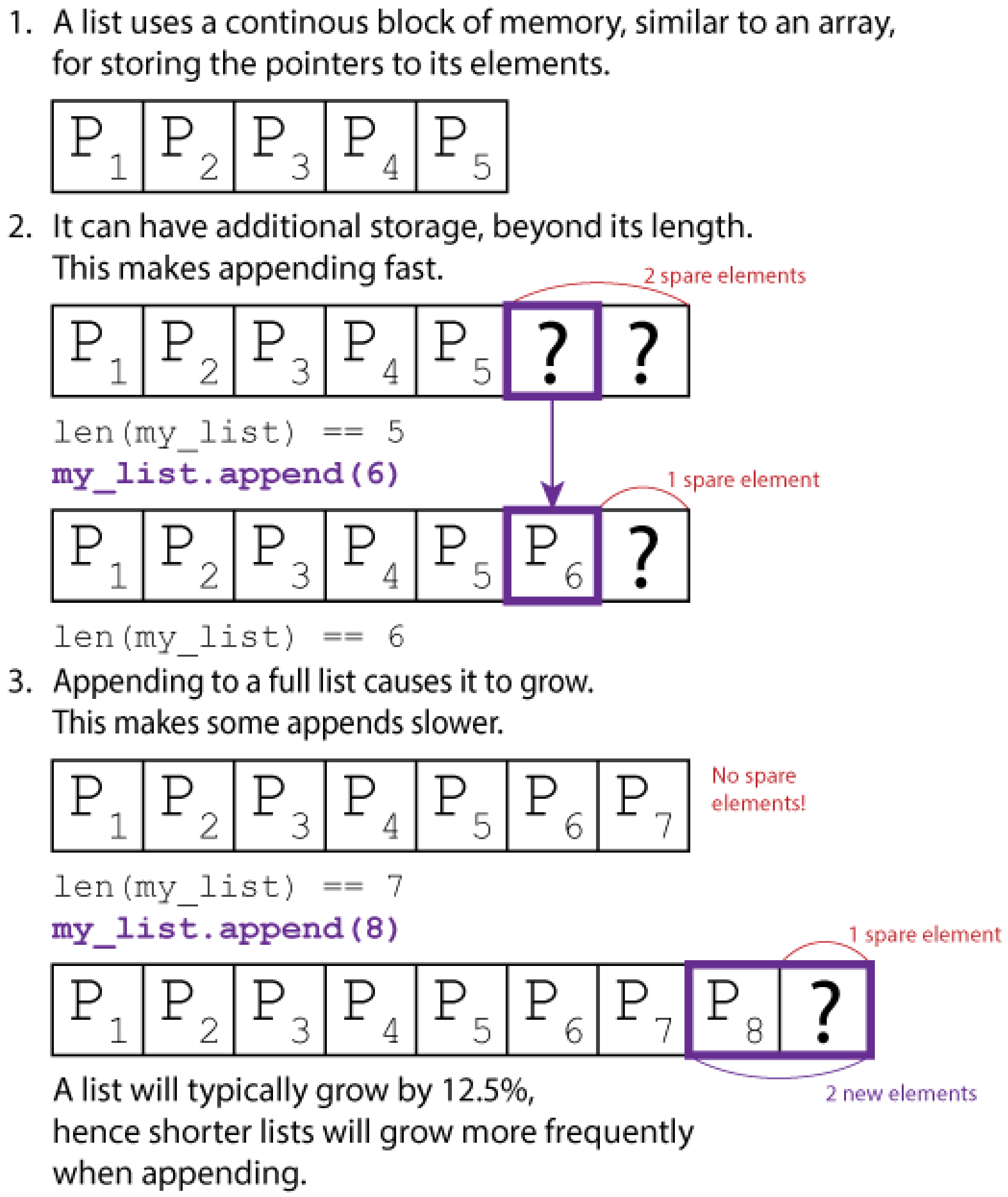
1. A list uses a continous block of memory, similar to an array, for storing the pointers to its elements.
P1 P2 P3 P4 P5
2. It can have additional storage, beyond its length. This makes appending fast.
2 spare elements
P1 P2 P3 P4 P5 ? ?
len(my_list) == 5
my_list.append(6)
1 spare element
P1 P2 P3 P4 P5 P6 ?
len(my_list) == 6
3. Appending to a full list causes it to grow. This makes some appends slower.
P1 P2 P3 P4 P5 P6 P7
No spare elements!
len(my_list) == 7
my_list.append(8)
1 spare element
P1 P2 P3 P4 P5 P6 P7 P8 ?
2 new elements
A list will typically grow by 12.5%, hence shorter lists will grow more frequently when appending.

Figure 1: How lists are typically implemented (not just in Python)

This has two implications:

- If you are growing a list with append(), there will be large amounts of redundant allocations and copies as the list grows.
- The resized list may use up to 10+% excess memory.

List Comprehension

If creating a list via append() is undesirable, the natural alternative is to use list-comprehension.

List comprehension can be twice as fast at building lists than using append(). This is primarily because list-comprehension allows Python to offload much of the computation into faster C code. General Python loops in contrast can be used for much more, so they remain in Python bytecode during computation which has additional overheads.

This can be demonstrated with the below benchmark:

```python
from timeit import timeit

def list_append():
    li = []
    for i in range(100000):
        li.append(i)

def list_preallocate():
    li = [0]*100000
    for i in range(100000):
        li[i] = i

def list_comprehension():
    li = [i for i in range(100000)]

repeats = 1000
print(f"Append: {timeit(list_append, number=repeats):.2f}ms")
print(f"Preallocate: {timeit(list_preallocate, number=repeats):.2f}ms")
print(f"Comprehension: {timeit(list_comprehension, number=repeats):.2f}ms")
```

Benchmarking results:

```
Append: 3.50ms
Preallocate: 2.48ms
Comprehension: 1.69ms
```

Let’s step back, and think about the techniques

- All the above general techniques contribute to overall code optimisation. By understanding the interplay between function calls, memory management, CPU caches, and code optimisation techniques, we **may** be able to create software programs that can run more efficiently and make better use of system resources, subject to the configuration of the working environment and some other uncontrollable factors.
- Benchmarking or profiling must be done to prove the optimisations done are effective.
- Do not optimise before the software has been proven correct.
- Abstractions not just hide the complex implementations, but also hide the performance implications.