1 Introduction

A fundamental principle in the development of computer systems is the concept of *multilevel abstraction*. This principle involves each level of abstraction building upon a lower level that provides more-detailed functionality. In essence, each level hides the intricate operations of the level below it. The use of multilevel abstractions forms a hierarchical stack that enables increasingly detailed operations.

At the top of this stack, specific human expressions, such as structured query language (SQL) in databases, are used to make processing requests. These requests then traverse through multiple levels of software, eventually reaching the bottom level where they are executed through detailed hardware operations. For instance, a final executable program, which relies on a machine-dependent instruction set architecture (ISA), is represented by a set of binary code. This binary code directly drives the hardware to execute each instruction of the upper-level abstraction.

Overall, the concept of multilevel abstraction allows for the systematic organization of computer systems, with higher levels of abstraction providing more user-friendly and human-expressive interfaces while relying on lower levels for efficient and detailed execution.

During program execution on a computer, the central processing unit (CPU) tends to access a small subset of data and instructions repeatedly in both time and space. This subset is commonly known as the *working set*. The principle of *temporal locality* refers to the frequent access of a particular set of data or instructions within a relatively short time frame. In simpler terms, if a dataset is accessed once, it is highly likely that it will be accessed again in the near future. On the other hand, the principle of *spatial locality* states that data access often follows a pattern where nearby data objects are accessed in sequence. In essence, if a particular data object is accessed, it is expected that the neighboring data objects will be accessed next.

These observations of common data access patterns form the basis of the principle of locality [45]. To leverage this principle, computer architects have designed a crucial hardware hierarchy known as *the memory hierarchy*. The goal of this hierarchy is to store frequently used data in close proximity to the CPU, ensuring low-latency access. Additionally, the memory and storage systems are designed to organize data in a sequential or sorted manner to optimize spatial locality.

In Figure 1.1, the memory hierarchy is depicted as a pyramid-shaped structure. Each level in the hierarchy is ranked based on its access latency and capacity, with both increasing in a top-down direction. The hierarchy begins with registers, which

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

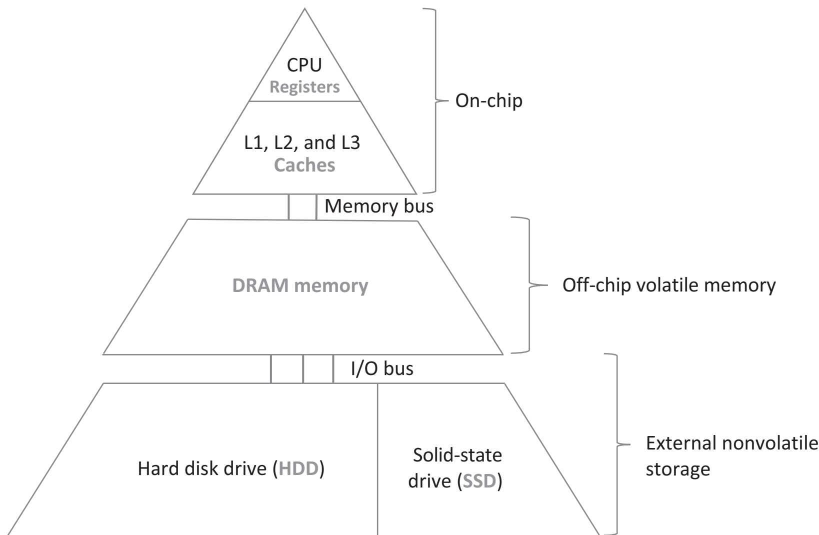

Figure 1.1 The memory hierarchy in computer architecture. At the top level of the hierarchy, we have the on-chip storage, which is located close to the CPU and includes registers as well as three levels of caches. Moving down the hierarchy, we encounter the middle level connected by a memory bus, which consists of off-chip DRAM memory. This type of memory is volatile, meaning it requires power to maintain the data that have already been stored. At the bottom level of the hierarchy, we have external storage devices that fall into the category of nonvolatile or persistent storage. These devices include HDD and/or SSD. They provide persistent storage for data and are not dependent on a continuous power supply to maintain the stored information.

are closest to the CPU, followed by on-chip caches spanning three levels. Next is the main memory, typically comprised of dynamic random-access memory (DRAM), and at the bottom level, we have persistent storage devices such as solid-state drives (SSD) and hard disk drives (HDD).

By implementing the memory hierarchy, computer systems can effectively exploit the principle of locality, optimizing data access patterns and improving overall system performance.

The implementation of software abstractions at multiple levels aims to enhance the productivity of software development by allowing developers to focus on each level without concerning themselves with the details of the subsequent levels. On the other hand, the hardware memory hierarchy is designed to reduce the latency gap between CPU-cycle-based computing and the growing volume of data movement.

The system design in multilevel abstraction and multilevel memory hierarchy in hardware are related and dependent on each other. Each layer of the hierarchy, including cache, main memory, and disk storage, has its own specific design and optimization considerations, which presents a challenge for achieving overall performance optimization across the entire system, particularly for different application execution patterns.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

In this chapter, we will delve into these two stacks: the software levels of abstractions and the hardware memory hierarchy. Our objective is to examine the interaction between these software and hardware stacks, as they play a crucial role in determining the performance of data processing. By understanding these relationships, we can gain valuable insights into optimizing system performance. We will also present several cases that illustrate the challenge of achieving overall performance, serving as motivation to comprehend the dynamic interactions among applications, architecture, and software systems. By examining these interactions, we aim to gain a deeper understanding of how they influence system performance and explore potential avenues for improvement.

1.1 The Multilevel Software Abstraction and the Deep Hardware Memory Hierarchy

Computing applications, regardless of their nature, rely on a computer architecture with software management support. In this architecture, two primary tasks are performed: (1) executing arithmetic and logic operations, and (2) accessing data stored in the extensive memory hierarchy. Figure 1.2 illustrates the combination of two hierarchies – the multilevel software abstractions and the underlying hardware architecture, with a particular emphasis on the pivotal role played by the memory hierarchy.

Within Figure 1.2, the software stack is interconnected with the hardware stack through a machine-dependent ISA, which represents the final level of software abstraction.

1.1.1 Software Abstractions

In Figure 1.2, the highest level of software abstraction is represented by "executable apps." These apps provide a simplified interface to users, concealing intricate execution details related to both software and hardware. Each app is identified by a unique icon and application name, enabling users to easily execute them simply by tapping on the icon. This level of abstraction benefits billions of people who do not possess extensive computing backgrounds or knowledge.

The subsequent level of abstraction is referred to as the "user domain." Within this domain, users can write programs using high-level programming languages like Java. Various programming tools are available in this domain, including debuggers for tracing, testing, and correcting programs, compilers that translate machine-independent high-level code into machine-dependent assembly code, assemblers that convert assembly code to machine binary code, linkers that combine multiple object files into a single executable program, and more. The abstraction of executable apps builds upon the user working domain abstraction, as every app is developed through user programming.

To execute a program on a computer, well-prepared executable code in the user working domain requires resource allocations for CPU cycles and memory space.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

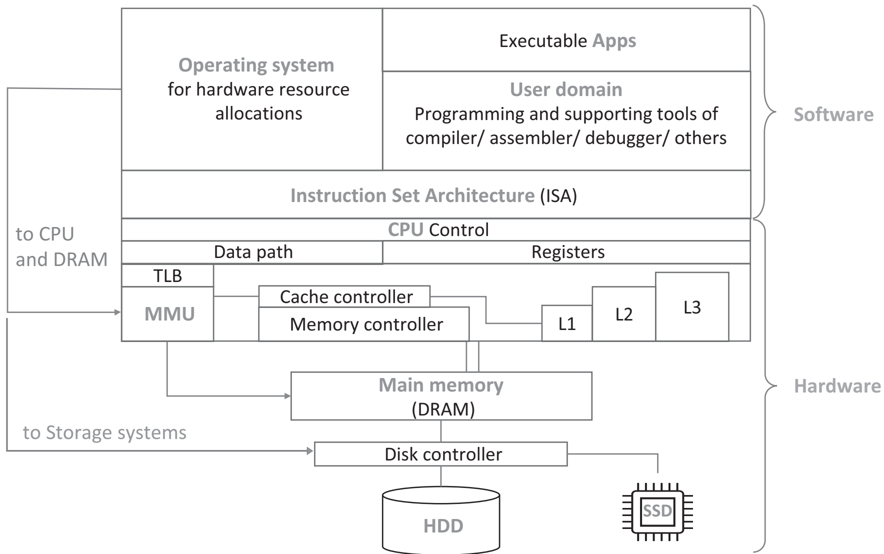

Figure 1.2 Key components in a general-purpose computer system. The highest level of abstraction in computing is the realm of executable apps, designed for billions of users to effortlessly run millions of applications without requiring any knowledge of the underlying computing system. Users simply need to touch the selected app icon on their screen using their finger. Moving down the hierarchy, we reach the programming environment. This level is developed for programmers, offering a variety of programming languages and tools that allow them to write programs. These programs can then be converted into assembly code and binary code, which are understood by the underlying computer hardware. The ISA forms the foundation for this assembly and binary code and is specific to the hardware platform. The OS plays a crucial role in the hierarchy by managing and allocating hardware resources to each executable program. It handles tasks such as scheduling CPU resources for computations, allocating memory space for data and instructions, and facilitating interactions with disk drives for reading and writing data. All of these components, including the OS, programming environment, and executable apps, reside within the software stack. The ISA serves as the crucial interface between the software and hardware layers, enabling communication and interaction between the two. The hardware layer provides a more-detailed illustration of the memory hierarchy, as depicted in Figure 1.1.

This allocation is facilitated by the operating system (OS), which acts as an agent between application programs and the hardware. The OS operates at another level of abstraction. It establishes critical connections with the computer hardware, specifically with the CPU, memory, and the storage system.

1.1.2 The Memory Hierarchy in Architecture

Within the hardware component depicted in Figure 1.2, a significant element is the memory hierarchy. However, there exists a fundamental distinction in the management approach between the OS and hardware architecture, primarily related to flexibility.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

In the realm of the OS, management policies are designed and implemented in software, allowing for a wide range of variations and ideas. The OS has the flexibility to employ diverse algorithms and policies to optimize memory utilization and performance. For instance, the OS dynamically allocates data pages in DRAM and employs sophisticated algorithms to select victim pages for eviction when the memory space becomes full. These algorithms are often more complex and adaptive than the predetermined mapping schemes employed by the hardware. The flexibility in OS management stems from the dynamic nature of the software and the ability to adapt to varying workload conditions.

In contrast, the internal management of the hardware is predominantly predetermined and operates through automatic processes. For example, the hardware swiftly identifies the cache block associated with a given memory address using cache designs like set-associative caches, with limited choices available. Similarly, memory pages are automatically mapped to specific memory banks following predefined interleaving rules. These predetermined hardware mechanisms are designed to ensure efficient execution in the critical path of the system, where even minor increments in latency can significantly degrade overall performance.

The hardware's reliance on predetermined management processes is rooted in the need to maintain precise timing and minimize delays in critical operations. Altering these processes in real time can be challenging and may introduce unpredictable performance consequences. Therefore, the hardware often operates with fixed and automatic management procedures to ensure predictable and efficient execution.

Overall, while the OS enjoys flexibility in managing memory through dynamic allocation and advanced algorithms, the hardware implements predetermined mechanisms optimized for speed and efficiency, prioritizing strict timing requirements in critical system operations.

1.2 Interactions among Software Abstractions and the Memory Hierarchy

The concept of abstraction and its implementation form the foundation of software and hardware development, offering significant benefits to the computing ecosystem in terms of efficiency and productivity. However, achieving effective interactions and information sharing among different domains within existing computing systems, both in software and hardware, is often hindered by a lack of interoperability.

Interoperability refers to the capability of diverse computer systems and software to exchange and utilize information seamlessly across independent domains. Unfortunately, current computing systems suffer from limited interoperability, resulting in constrained interactions and information sharing between the abstractions and the memory hierarchy.

As a consequence, the full potential of the computing ecosystem remains untapped, as effective interoperability facilitates efficient data exchange and collaboration among different software and hardware components. Enhancing interoperability can unlock new possibilities for innovation, enabling more efficient and seamless interactions between software abstractions and the memory hierarchy.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

1.2.1 The Merits of Multilevel Abstractions

The use of software abstractions is a widespread practice in the advancement of computing infrastructure across various applications. While Figure 1.2 illustrates some basic levels of abstraction, it is important to note that there are additional examples, such as remote procedure call (RPC) [19]. Remote procedure call offers users an abstraction that allows them to utilize computing services on a remote node, effectively hiding the intricate networking details. This abstraction provides users with a simplified interface, resembling a local procedure call, while transparently handling the complexities of remote execution.

The introduction of multilevel abstractions has significantly enhanced software development productivity. Different programming tasks can now be specialized, with focused attention given to specific computing domains. For instance, at the user level, developers can analyze and build application programs, while at the OS kernel level, they can work on system development. Similarly, networking programming can be accomplished by utilizing communication protocols, among other specialized tasks. Each level of abstraction defines a distinct computing domain, allowing programmers to concentrate on their specific tasks without being burdened by the details of lower levels.

This design approach aligns with the fundamental economic principle of "Division of Labor" or "Specialization of the Labor Force." By utilizing multilevel abstractions, a broad range of users in various domain areas can efficiently develop and run their programs on computers. This specialization not only enhances productivity but also enables users to leverage computing resources effectively within their respective domains.

> Economic growth is rooted in the increasing *division of labor*, which is primarily related to the *specialization of the labor force*, essentially the breaking down of large jobs into many tiny components. Each worker becomes an expert in one isolated area of production, increasing his efficiency.
>
> Adam Smith, *The Wealth of Nations*, 1776 (our emphasis).

Level-focused software development has historically demonstrated high efficiency, primarily due to the continuous performance improvements at the circuit level, as dictated by Moore's Law. This improvement has benefited all levels of abstraction within the computing ecosystem. Consequently, the performance loss resulting from inefficient interactions between levels and the underlying architecture has not been a significant concern.

However, as we approach the end of Moore's Law, the challenge of sustaining and further enhancing the performance of computing infrastructure to meet the increasingly high demands of various applications, particularly data-centric ones, has become a significant hurdle.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

Data-centric applications, which rely heavily on processing large volumes of data, pose unique challenges in terms of performance. The traditional level-focused development approach may not be sufficient to address these challenges. Inefficient interactions and architectural limitations can become bottlenecks that impede the performance gains required for these demanding applications.

As a result, finding innovative solutions to continue raising the performance of computing infrastructure has become a major challenge for the industry. It requires exploring new avenues beyond traditional approaches and considering alternative strategies to optimize performance and address the demands of data-centric applications.

Efforts are being made to explore novel architectural designs, develop specialized hardware accelerators, leverage parallel and distributed computing paradigms, and invest in advanced memory and storage technologies. Additionally, optimizing software algorithms and leveraging machine learning techniques can also contribute to performance improvements.

Addressing these challenges necessitates a collaborative effort from researchers, engineers, and the computing community as a whole. By exploring new technologies, refining existing approaches, and embracing interdisciplinary solutions, we can tackle the performance demands posed by data-centric applications in a post-Moore's Law era.

1.2.2 Balancing Multiple Performance Objectives

The majority of programming tasks are carried out in the user working domain (as depicted in Figure 1.2), which operates at a logical and machine-independent level. Algorithmic complexity plays a crucial role in achieving high performance and is often represented using the Big-O notation, which quantifies the number of computing operations as a function of the data size. For instance, the bubble sort algorithm has a complexity of $O(n^2)$, where n represents the number of elements to be sorted. In the context of Big-O notation, algorithmic complexity assumes that data movement is cost free, with computing operations being the primary performance determinant.

However, in today's computing landscape, achieving highly efficient program execution relies on three critical factors: (1) attaining low algorithmic complexity, (2) minimizing data movement within the memory hierarchy, and (3) maximizing parallelism in both software and hardware. Table 1.1 characterizes the dynamics of program execution influenced by these three factors. The first row consists of all zeros, representing the rare scenario where none of the factors favor the program. Conversely, the last row contains all ones, indicating that all three factors align favorably for the program. These two extreme cases are infrequent in practical scenarios.

Below the row of three 0s, Table 1.1 comprises a set of three rows, each highlighting a single positive factor (value 1). However, a program's performance may still be uncertain in these cases, as relying solely on a single factor might not

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

Table 1.1 The performance of a program during execution is determined by three key factors: (1) algorithmic complexity, (2) data locality, and (3) parallelism. These factors interact to produce eight possible combinations of values. A value of 0 indicates that the program is not favored by that particular factor, representing high complexity, low locality, or low parallelism, respectively. Conversely, a value of 1 indicates that the program benefits from the factor, representing low complexity, high locality, or high parallelism, respectively.

Notes	Parallelism	Data locality	Complexity
Inefficient execution	0	0	0
Uncertain performance	0	0	1
	0	1	0
	1	0	1
Perfect sequential	0	1	1
Low locality	1	0	1
Possible high performance	1	1	0
Perfect execution	1	1	1

be sufficient for achieving acceptable execution efficiency. Additionally, there are three rows in Table 1.1 where two positive factors are present. Let us examine each case.

The row denoted by 0, 1, 1 represents a perfect sequential execution. In this case, the program parallelism is low (= 0), but both data locality (= 1) and complexity (= 1) are favorable.

The row of 1, 0, 1 gives another uncertain case. When parallelism is high (= 1) and algorithmic complexity is low (= 1), but data locality is poor (= 0), the question arises as to whether the benefits of high parallelism and low complexity can offset the overhead caused by data movement. Consequently, the program's performance may also be uncertain in such scenarios.

For the row of 1, 1, 0, despite the high complexity, studies have demonstrated that high-performance execution is still feasible in these scenarios due to the high strengths of both locality and parallelism [52].

In summary, while algorithmic complexity remains an important consideration, achieving efficient program execution in today's reality relies on a careful balance between low complexity, minimized data movement, and maximized parallelism. Various combinations of these factors can impact performance, and their interplay must be carefully evaluated to determine the most effective approach for a given program.

Considering the dynamic nature of execution performance influenced by the three factors, it becomes imperative for us to address effective interactions among software abstractions and the memory hierarchy. These interactions are crucial for sustaining high efficiency and performance during application execution. It is the responsibility of system and architecture professionals to tackle this challenge, alleviating the burden on application users. In Section 1.3, we will provide a concise overview of several case studies that highlight the importance of these interactions.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

1.3 Case Studies: Performance Impact of Interaction Optimizations

This section highlights two data processing case studies that demonstrate the limitations of relying solely on user-level abstractions for efficient execution. It underscores the importance of interactive design between software and hardware to achieve high performance.

> What makes a programmer a good one, is mostly the ability to shift levels of abstraction, from low level to high level, to see something in the small and to see something in the large.
>
> A quote from Donald Knuth, *Dr. Dobb's Journal*, 1996.

1.3.1 Optimizations for Matrix Multiplication by Crossing Abstraction Levels

Matrix multiplication computations are extensively employed in machine learning and various scientific applications. In a paper [90] published in *Science*, the authors present a compelling case study showcasing the significant reduction in execution time for a 4096×4096 matrix multiplication. Through a series of concerted efforts involving effective interactions among software abstractions and leveraging parallelism and locality within the hardware architecture, the execution time was improved by an astounding factor of approximately 63,000. Initially, the program was written in Python, as shown here.

```
for i in xrange(4096):
  for j in xrange(4096):
     for k in xrange(4096):
        C[i][j] += A[i][k] * B[k][j]
```

These four lines of Python code have undergone a series of optimizations, building upon each previous implementation and resulting in cumulative performance improvements. The first optimization involves reimplementing the code in Java, a more-efficient programming language. This change alone leads to a reduction in execution time by a factor of 10.8. Subsequently, the matrix multiplication is implemented in C, a high-level language primarily designed for system programming. This modification further reduces the execution time by an additional factor of 4.4. These two optimizations, achieved through different programming language choices, operate at the logical level of abstraction and result in a remarkable 47-fold speedup compared to the original Python code.

The subsequent four optimizations involve cross-level interactions, harnessing the capabilities of the hardware. The first optimization involves parallelizing the loop using multiple cores (18 cores in this case). The second optimization exploits the memory hierarchy locality through a divide and conquer implementation. The third optimization focuses on vectorizing the parallel program. The final optimization

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

Table 1.2 A sequence of optimizations for matrix-multiplication of 4096 by crossing abstraction levels. "GFLOPS" is the billions of 64-bit floating-point operations per second. The baseline program is the Python implementation, and the speedup numbers are relative to that of the baseline program. The performance numbers in this table are quoted from [90].

Implementation	Execution time (s)	GFLOPS	Speedup
Python	25,552.48	0.005	1
Java	2,372.68	0.058	11
C	542.67	0.253	47
Parallel Loops	69.80	1.969	366
Divide and Conquer	3.8	36.180	6,727
Vectorization	1.10	124.914	23,224
AVX instructions	0.41	337.812	62,806

utilizes Intel's Advanced Vector Extensions (AVX) instructions, enabling direct interaction with the hardware architecture. Collectively, these four optimizations that leverage hardware features significantly reduce the execution time by a factor of 1300 compared to the optimized sequential C program. Table 1.2 illustrates the step-by-step reductions in execution time achieved through each optimization.

1.3.2 Sorting Algorithms and the Memory Performance

Sorting algorithms play a fundamental role in various data processing applications and are frequently utilized. Traditional algorithm analysis primarily focuses on counting the number of logical comparison operations performed during sorting to differentiate the complexities of different sorting algorithms. For instance, bubble sort requires n^2 comparisons, while merge sort requires $n \log(n)$ comparisons, where n represents the number of elements being sorted. However, in practice, the execution time for comparisons is significantly shorter compared to the time spent on data movement during sorting operations.

In particular, data movement refers to the loading of data from the DRAM memory to on-chip caches, assuming that all sorting operations are performed in memory. Researchers have recognized the importance of counting the number of memory requests and the amount of data loaded from memory during the execution of a sorting algorithm. These metrics have become integral components of algorithm analysis for sorting algorithms, as they shed light on the data movement and its impact on the overall performance.

The unconventional approach of analyzing algorithms based on data movement between memory and caches has gained increasing significance due to the dominant role of data-movement operations in overall execution time. It provides valuable insights into tailoring the design of sorting algorithms to align with the underlying cache organization, aiming to maximize cache hits by minimizing memory requests. This approach, often referred to as "algorithms design for memory hierarchies" [99],

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

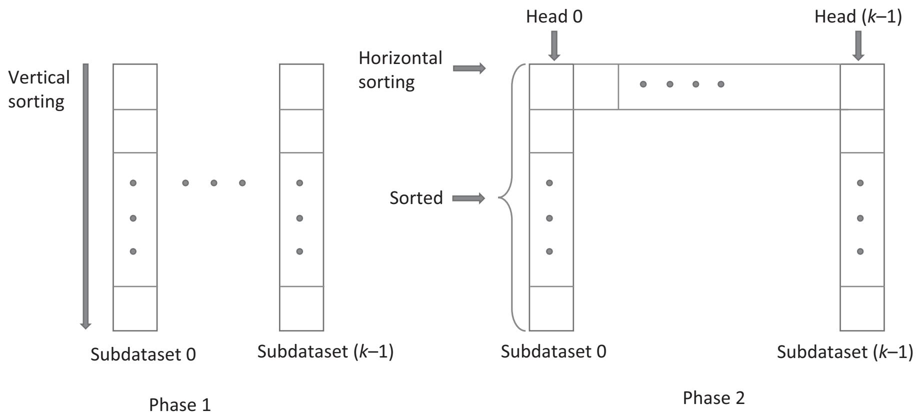

Figure 1.3 Two phases in a tiled merge sort: Each of the k vertical sorting in phase 1 is done in cache, and the iterative horizontal sorting in phase 2 is also in cache.

is dependent on the specific characteristics of the cache architecture. By considering the cache hierarchy during algorithm design, developers can optimize the utilization of caches and improve overall performance.

We will focus on implementing sequential sorting algorithms within a single node, using merge sort as our example. One technique we will explore is tiled merge sort, introduced in [83]. Tiled merge sort involves partitioning the dataset into multiple subdatasets, denoted by k. The key idea behind this technique is to process two subdatasets at a time, assuming they can fit into the cache. Each subdataset is approximately half the size of the cache or smaller. By sorting each subset individually, we can effectively avoid cache capacity misses and make full use of the data loaded in the cache without interference. This allows us to perform the entire sorting process in the cache, incurring only compulsory (cold) misses.

The restructured merge sort algorithm consists of two phases. In the first phase (phase 1 in Figure 1.3), each subdataset is sorted vertically using the basic merge-sort algorithm. After sorting, each subset has a head pointer pointing to the smallest number within it.

In the second phase (phase 2 in Figure 1.3), a k-way merge method is employed to horizontally merge all the sorted subdatasets. This is achieved by creating a horizontal array that consists of the head elements of the sorted subdatasets. The horizontal array is then sorted. Similar to the first phase, the size of the horizontal array is chosen to be approximately half or smaller than that of the cache size. At the end of each horizontal array sort, the subdataset containing the selected winner contributes its next element from the head to the array. This iterative process continues until the last element (the largest number) in the last horizontal array is selected. A tournament sort [97] can be used within each horizontal merge sort. The tournament sort, also known as single-elimination tournament, is often used to select the winner in sports competitions and elections.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

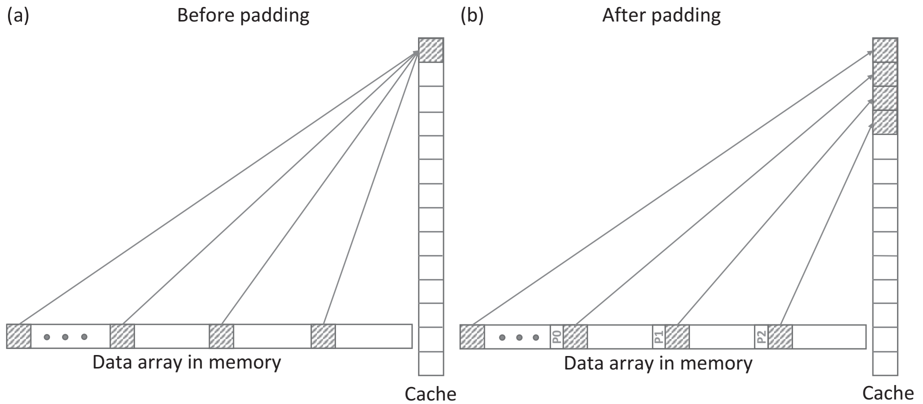

Figure 1.4 Effect of padding: (a) Before padding, a set of data elements maps to the same cache location; (b) after padding, the virtual memory addresses for the data elements are shifted by padding element p_i for $i = 1$ to $k - 1$, so that the same conflicting data elements are mapped to different cache locations.

Tiled merge sort ensures that cache capacity misses are avoided in each sorting phase. The reason for utilizing half the cache size for data is to allocate additional space for sorting-related operations, such as building a tournament tree in the second phase of horizontal sorting.

Cache capacity misses can be mitigated by partitioning the dataset, but addressing cache conflict misses is more complex. Cache conflict misses occur when multiple memory addresses map to the same cache location, resulting in cache misses. This is because the cache size is only a small portion of the main memory. Thus, only a small number of memory address bits is used for cache mapping. We will have more discussions on this issue in Chapter 3. One effective approach to tackle this issue is to modify the physical data layout to shift memory addresses. This can be accomplished at runtime by system software [17, 138]. Additionally, users can modify the data layout at the user level by inserting extra elements in specific locations within the dataset. By altering the virtual memory addresses of data elements, the base addresses of potentially conflicting cache locations are also changed. This technique is referred to as "padding" and has been successfully applied in the design of sorting algorithms [137].

Figure 1.4(a) illustrates an example of conflict mapping where multiple memory addresses of data elements are mapped to the same cache location in a direct-map cache (before padding). After applying padding, as shown in Figure 1.4(b), these conflicting memory addresses are mapped to different cache locations, effectively reducing cache conflict misses.

1.4 The PAPA Concept for Computing Interactions

The process of transforming a computing task, starting from its problem formulation by a human and ending with its execution in a machine, involves dynamic interactions

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

among four key components or steps. We refer these components as PAPA, which stands for Premier, Algorithm, Program, and Advancement.

- *Premier* In computational problem solving, the Premier serves as a pivotal concept or methodological principle, steering the design and implementation of the computational process towards a viable solution. Unlike elementary arithmetic problems such as $1 + 1 = 2$, in practice, myriad problems do not readily decompose into a sequence of computational steps executable by a computer, a complexity that intensifies when parallel computing is employed. Therefore, Premier's foundational role is to "facilitate feasibility." It aligns more intimately with the domain knowledge or mathematical principles underlying the target problem rather than with the specifications of any particular computing system. For instance, consider the problem of calculating the area of a triangle given the coordinates of its three vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3). A Premier-guided solution would involve employing the following formula: $area = \frac{1}{2} \cdot abs(x_1 \cdot (y_2 - y_3) + x_2 \cdot (y_3 - y_1) + x_3 \cdot (y_1 - y_2))$. This illustrates how the Premier component remains independent of the specific machine or computing system in use, focusing instead on the fundamental principles guiding the solution process.
- *Algorithm* An algorithm embodies a computational procedure characterized by a sequence of discrete steps or fundamental operations, each readily executable by a computer. Illustratively, in the aforementioned example of calculating the area of a triangle, the algorithm encompasses the systematic steps required to evaluate the right-hand side of the formula, involving a succession of elementary arithmetic computations. A vital aspect characterizing an algorithm is its complexity, which can encompass various dimensions such as time complexity, space complexity, input/output (I/O) complexity, and communication complexity. Among these, time complexity is the most important one that assesses the total number of computing operations (a function of the input size, e.g., O(1) for the above example or $O(n^2)$ for bubble sort) required by the algorithm, offering insights into potential execution durations. Another pivotal concept within the realm of algorithms is recurrence, which refers to the repetitive occurrence or repetition of a specific operational pattern within the algorithm. This concept is prevalently employed within recursive algorithms, wherein a problem is segmented into smaller subproblems of identical nature, facilitating more manageable solutions. Notably, the algorithm maintains its independence from the underlying machine or computing system. Its design and attributes are not confined to any particular hardware configuration, thereby enabling implementation across diverse platforms.
- *Program* This component embodies the concrete implementation of the algorithm. Its design is shaped by various factors including the programming environment, system support, and the particularities of the machine or computing system in use. Consequently, the program can exhibit a high degree of dependency on these elements. Notably, even when based on the same algorithm, a program scripted in a high-level language (such as Python) can markedly differ in appearance from one crafted in assembly language. Illustratively, the following Python code, used in the preceding example, mirrors the original formula quite closely.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

```
x1, y1 = 0, 0
x2, y2 = 5, 0
x3, y3 = 0, 4
area = abs(x1*(y2 - y3) + x2*(y3 - y1) + x3*(y1 - y2)) / 2
```

However, when utilizing a lower-level programming language, a myriad of intricate and complex programming concepts become more prominent, encompassing aspects such as memory management and register allocation (e.g., determining the storage location of a variable). The subsequent code showcases the same process, articulated in ARMv7 assembly language.

```
.data
x1: .word 0
y1: .word 0
x2: .word 5
y2: .word 0
x3: .word 0
y3: .word 4
area: .word 0

.text
.global _start
_start:
    LDR R0, =x1
    LDR R1, =y2
    LDR R2, =y3
    LDR R3, [R0]
    LDR R4, [R1]
    LDR R5, [R2]

    SUB R6, R4, R5
    MUL R6, R3, R6

    LDR R0, =x2
    LDR R3, [R0]
    SUB R4, R5, R1
    MUL R4, R3, R4

    ADD R6, R6, R4

    LDR R0, =x3
    LDR R3, [R0]
    LDR R1, =y1
    LDR R4, [R1]
    SUB R5, R4, R2
```

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

```
MUL R5, R3, R5

ADD R6, R6, R5
MOV R7, #2
SDIV R6, R6, R7

STR R6, =area
```

For readers unfamiliar with assembly languages, the aforementioned R0-R7 denote CPU registers, while LDR/STR signify load/store instructions, and ADD/SUB/MUL/SDIV stand as arithmetic instructions.

- *Advancement* This focuses on optimizing the mapping of the Algorithm and Program to the unique hardware features of a machine as well as addressing potential corner cases during the Algorithm and Program phases. The Advancement is dependent on the machine and its system software. The final performance of any computing task is determined by interactive optimization methods that take into account the interplay between the Algorithm, its Program and the underlying software and hardware systems. This optimization process aims to improve the overall efficiency and effectiveness of the program's execution. It involves fine tuning various aspects, such as code optimization, resource allocation, and system configuration, to achieve optimal performance. For example, in the aforementioned example, we can find that the computation process for the area calculation actually contains three parts that have the same operations but only different inputs: $(x1*(y2-y3))$, $x2*(y3-y1)$, $x3*(y1-y2))$. Therefore, instead of executing them one by one as the previous assembly language code shows, we can use advanced vector instructions offed by the modern CPU to batch execute the three parts in a SIMD way (*single-instruction, multiple-data*). As we will introduce in later chapters for graphic processing unit (GPU) programming, we can see that such an Advancement step can be very difficult due to the complicated architecture of advanced computer hardware.

The PAPA process embraces the potential for iterative interactions across its four stages, wherein several cycles of redesign and implementation may be requisite. Generally, the PAPA process represents a multidisciplinary endeavor aimed at optimally solving practical problems through the utilization of modern hardware. This approach melds foundational mathematics, domain-specific knowledge, algorithm analysis, software engineering, and architectural optimizations into a cohesive strategy.

1.5 Summary

In this chapter, we have provided a brief overview of the interactions between software abstractions and the memory hierarchy in computer architecture. Achieving highly efficient program execution requires finding a balance among three critical factors: low algorithm complexity, low data movement, and high parallelism. To further enhance

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002

the performance of computing and data processing, it is essential to establish effective interactions among the virtual space of the user domain, the memory space of the operating system and architecture domains, and the storage space of external devices in both software and hardware.

These interactions often involve disruptive changes that may require adding, modifying, or bypassing conventional abstractions. As computer scientists, it is not practical to expect application users to be deeply involved in these long-term developments. Instead, our role is to provide tools and frameworks that enable users to automatically benefit from performance improvements in restructured ecosystems. For instance, ATLAS [131] is a software tool designed to automatically determine cache parameters through extensive testing of linear algebra algorithms during runtime. By automating the optimization process, tools like ATLAS would help users to effortlessly enhance the performance of their applications without requiring extensive manual intervention. By focusing on developing such tools and frameworks, we can pave the way for a more efficient and productive computing environment, where users can harness the benefits of effective interactions without the need for specialized expertise.

At the end of the chapter, we presented the concept of PAPA as a comprehensive framework encompassing the entire life cycle of any computing tasks. The PAPA concept and its implementation at various levels serve as central themes throughout the book.

Downloaded from https://www.cambridge.org/core. IP address: 3.141.38.172, on 27 Feb 2025 at 06:41:15, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009122115.002