1
Introduction

Welcome to the study of advanced computer architecture. The first part [Sarangi, 2015] of this two-book series explains the basic concepts of computer organisation and architecture. In the first part, we discussed the basics of computer architecture: instruction sets (*SimpleRisc*, ARM, and x86), processor design (gates, memories, processor, pipeline), the memory system, multiprocessor systems, and I/O systems. We started from scratch and left the reader at a point where she could understand the essentials of a computer architecture, write simple assembly programs, understand the intricacies of pipelines, and appreciate the nuances of memory and I/O system design. Additionally, we also provided a foundation for understanding multiprocessor systems, which is the bedrock of a course on advanced computer architecture.

Unfortunately, the processor described in a basic course on computer architecture is hardly used today other than in some extremely simple and rudimentary devices. A processor used in a modern device starting from a smart watch to a server uses far more sophisticated techniques. These techniques are typically not covered in a basic course on computer architecture. Hence, we shall discuss such techniques in this book, and cover them in exquisite detail.

Let us start with understanding and appreciating the fact that the modern processor is not just a sophisticated pipeline. The pipeline is supplied data from the memory system. Unless the memory system is efficient and can provide high bandwidth, we cannot run a high performance pipeline. Moreover, modern processors do not necessarily have a single pipeline. They have multiple pipelines, where each pipeline along with its caches is known as a *core*. We thus are in the era of multi-core processors, where a processor chip has a multitude of cores, caches, and an elaborate network that connects them. In addition, this ensemble needs to be power-efficient, secure, and reliable.

Let us motivate our discussion further by considering the basic drivers of technological change in the world of computer architecture. A processor sits between the hardware and the software. Its aim is to leverage the features provided by the latest hardware technology to run software as efficiently as possible. As the underlying hardware keeps on improving, it becomes imperative to modify the computer architecture to exploit these improvements. The most important empirical law in this space is known as the *Moore's law*. It was proposed by Gordon Moore, the co-founder of Intel, in 1965. He postulated that because of advances in transistor technology, the number of transistors per chip will double roughly every year. He has been extremely prescient in his observation. Since 1965 the number of transistors per chip have doubled roughly every 1-2 years (see Figure 1.1). The processor industry went from generation to generation where the size of transistors decreased by a factor of $\sqrt{2}$. This ensured that the number of transistors per unit area doubled every generation. Till 2010 these transistors were being used to create more sophisticated processors and increase the on-chip memory. However, after 2010 the extra transistors are being used to increase the number of processors (cores) per chip mainly because the gains from increasing the complexity of a core are limited, and high power dissipation is a very major issue.

Off late, Moore's law is showing signs of saturation. Current feature sizes (smallest feature that can be fabricated) are at 7 nm as of 2020, and can only decrease till 5 nm. Beyond that, the size of a feature on silicon will become too small to fabricate. A 5 nm structure is only 25 silicon atoms wide! The trends seem to indicate that we will move to more application specific processors that will solve specific problems from different domains, particularly from the machine learning domain. Subsequently we need to move to non-silicon technologies.

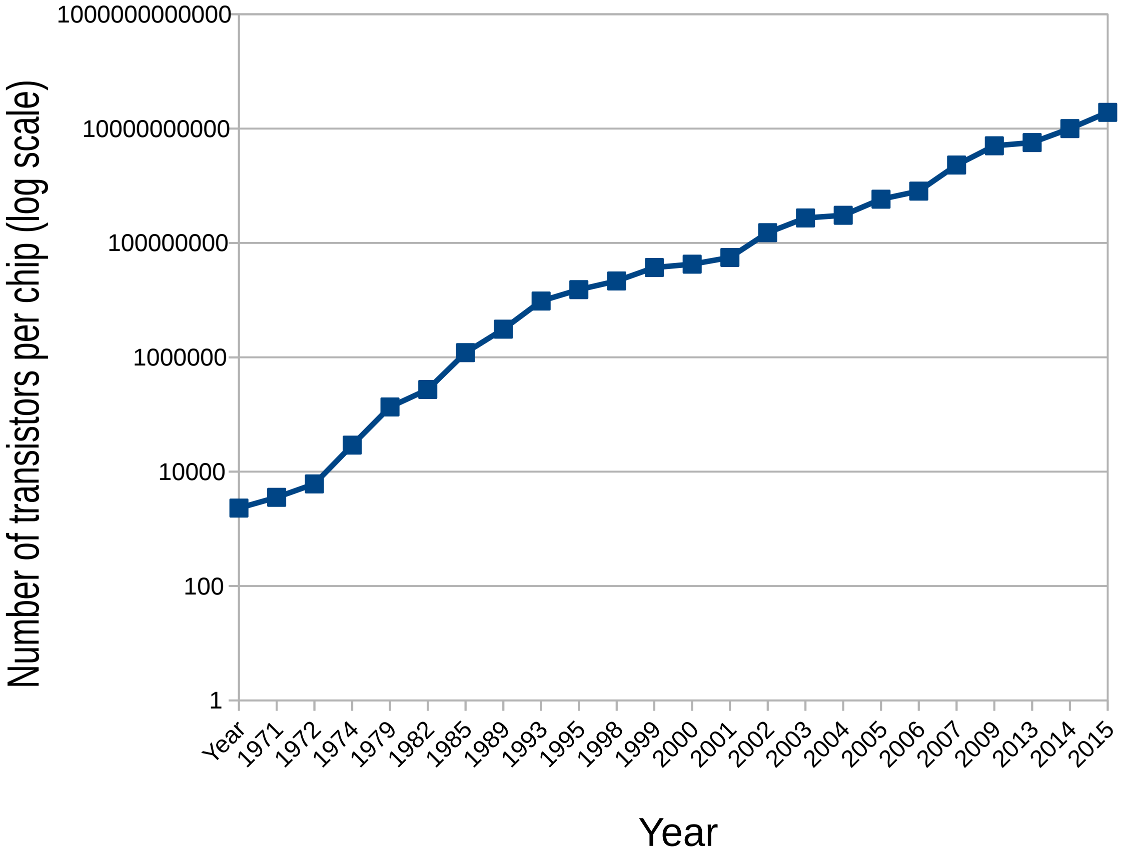

Figure 1.1: Transistors per chip over the last 50 years (adapted with modifications, source: [Rupp, 2017], licensed and distributed under the Creative Commons ShareAlike 4.0 license [ccs,])

Definition 1
It is an empirical observation that the number of transistors in a chip doubles every 1-2 years. Ever since it was proposed by Gordon Moore in 1965, it continued to hold till roughly 2012. Henceforth, the rate of technological change slowed down.

Most semiconductor companies move from one technology generation to another, where the feature size reduces by a factor of $\sqrt{2}$ every generation. The feature size *is defined as the dimensions of the* *smallest possible structure that can be reliably fabricated on silicon.*

In 1974 another empirical law was proposed by Robert Dennard, which is known as *Dennard scaling*. Dennard postulated that the performance per Watt also grows exponentially (tracks Moore's law). This will only happen if we can increase the frequency, and the total number of instructions processed per cycle every generation. This has ceased to happen since 2006 mainly because the power dissipation of chips outpaced gains in performance.

Let us now go through the organisation of the book keeping in mind that till roughly 2010 our main aim was to take advantage of Moore's law and Dennard scaling. The goal posts have changed henceforth.

1.1 Moving from In-order to Out-of-order Pipelines

A basic course on computer architecture talks about a simple 5-stage in-order pipeline (see Section 2.1 for more explanation). An in-order pipeline is a very basic pipeline where instructions are processed in program order (order in which they appear in the program). This is unfortunately an inefficient implementation. It is necessary to further complicate the processor.

A naive approach to do this is to issue multiple instructions in the same cycle. This augments our classic 5-stage pipeline that moved only one instruction from one stage to the next in a single cycle. In this case we move a bundle of k ($k = 2$, or $k = 4$) instructions from one stage to the next. This process does introduce complexities in the interlock and forwarding logic; however, for the time being let us assume that all of these are solved problems. Let us compare the increase in the instructions per cycle (IPC) for a standard set of SPEC benchmarks [Henning, 2006] with this technique. The SPEC benchmark suite is a standard set of programs that is used to evaluate the performance of processors.

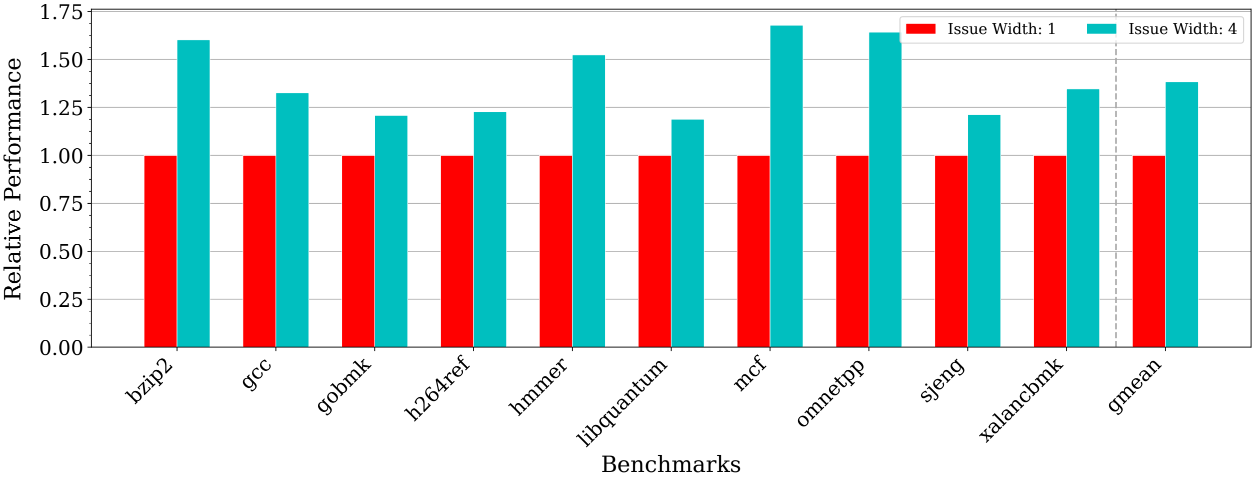

Figure 1.2: Performance comparison between 4-issue and 1-issue in-order processors (SPEC CPU$^{®}$ 2006 benchmarks(`https://www.spec.org`))

Figure 1.2 shows the comparison of the performance (inversely proportional to the simulated execution time) between a 4-issue and a 1-issue in-order pipeline. A 4-issue in-order pipeline creates bundles of at most four instructions and processes/executes all of them in one go. This requires four times the number of functional units. For example, if we want to execute four add instructions in parallel, we need four adders instead of just one adder as in a simple pipeline. We expect this pipeline to be much faster than traditional processors that we have all studied in a basic computer architecture course. The traditional pipeline is a 1-issue pipeline, where only one instruction is processed every cycle. However, as we see from Figure 1.2, this is not the case. The performance does not increase proportionately and it looks like our investment in adding more on-chip resources has not paid off. A 4-issue in-order pipeline just gives us 37.5% more performance on an average. Here, **performance** is defined as the **reciprocal of the execution time**. The results were generated using an architectural simulator (the Tejas Simulator^{TM}) that is discussed in Appendix B. An architectural simulator is a virtual processor that simulates the behaviour and timing of a real processor. It can be used to execute full programs (the SPEC CPU 2006 benchmarks in this case) and estimate their execution time and power. Additionally, we can get detailed statistics for each execution unit in the pipeline and in the memory system.

Let us thus move ahead and break the in-order assumption. This means that it is not necessary to execute instructions in program order. Consider the code shown in Figure 1.3. Assume that we can process/execute a maximum of two instructions per cycle. Given the fact that the code has strict dependences [1] between consecutive instructions, it is not possible to execute most pairs of consecutive instructions in parallel if we stick to in-order processors – executing them in program order. However, if we are allowed to pick instructions at will and execute them without affecting the correctness of the program, we could achieve a much higher performance. Such processors are known as out-of-order (OOO) processors. These processors do not execute instructions in program order; however, they respect all data and control dependences.

Figure 1.3 shows a code snippet written in the *SimpleRisc* assembly language (see Appendix A). The code with a 2-issue in-order pipeline takes 5 cycles, whereas, with an OOO processor, we need 3 cycles. This is a performance improvement of 40%.

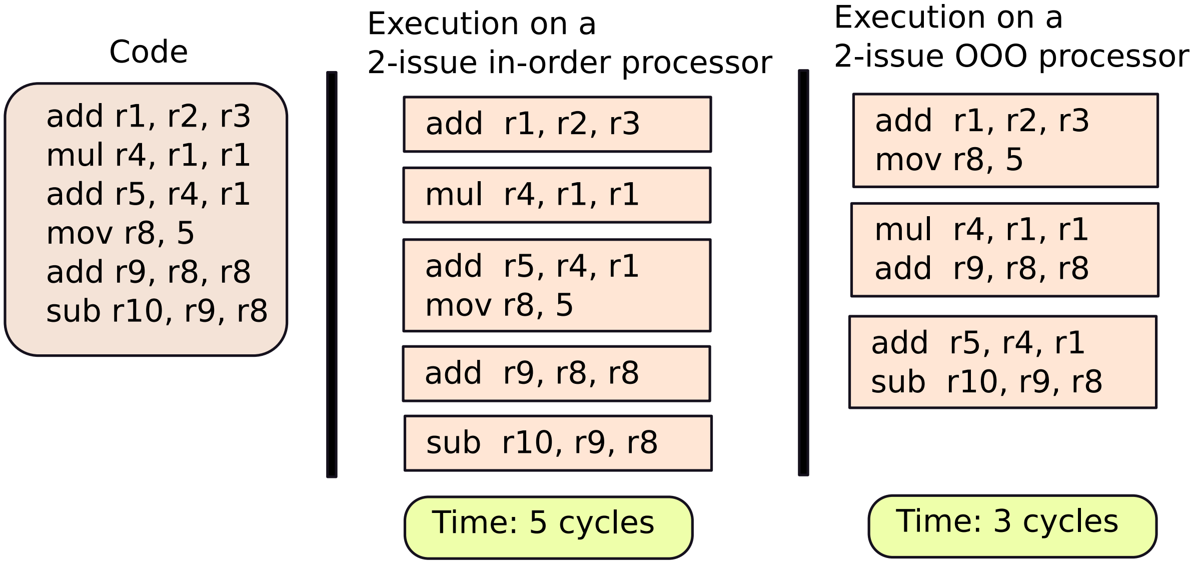

Figure 1.3: Execution on in-order and out-of-order processors

Clearly, in this case an OOO processor is the winner. To minimise the execution time, we need to increase the number of instructions we can process per cycle. This is quantified as the IPC (instructions per cycle) metric. To maximise the IPC, having more choice is better, and in this case OOO processors offer a far greater degree of freedom with respect to choosing instructions for execution. However, this sounds simpler than it actually is. Intelligently choosing instructions and executing them in parallel is very difficult to do

[1] Note that there are two English words, “dependence” and “dependency” – both refer to the quality of being dependent. However, we shall consistently use the term “dependence” in this book because “dependency” typical refers to a geopolitical entity that is controlled by a higher power. The plural form will be “dependences”.

in practice. We need to ensure the correctness of the program by ensuring that no dependences are being violated, all the branches are resolved correctly, and nothing wrong is being written to memory. We devote 4 chapters to study the intricate mechanisms involved in this process. Ensuring that we can pick as many independent instructions as possible without violating any correctness constraints is difficult and requires a complete redesign of our basic processor.

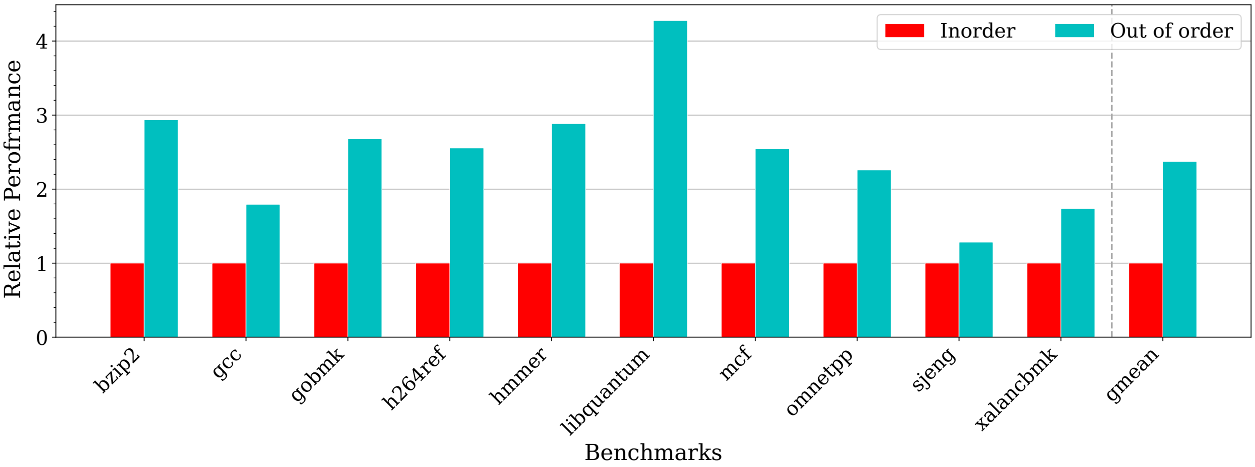

Figure 1.4: Performance comparison: in-order vs out-of-order processors

Figure 1.4 shows the comparison of the performance obtained for a 4-issue in-order pipeline and a 4-issue OOO pipeline. We observe that we get a mean performance improvement of 133% by using an OOO pipeline. This is significant in the world of processor design, and explains why most desktop and server processors today use OOO pipelines. We shall discuss OOO pipelines in Chapters 2, 3, 4, and 5.

1.2 Moving to Multicore Processors

Figure 1.5 shows the historical increase in processor (single core) performance over the last two decades. A *core* is defined as a single processing unit that has one pipeline and possibly a set of caches (data cache and instruction cache).

As we can see, from 2009 onwards, the processor (single core) performance has saturated. Due to limitations imposed by power, temperature, and complexity, and sometimes the available amount of instruction level parallelism in the code, the performance could not increase further. Given the fact that investing on a single core produced little returns, we needed to move towards multicore processors, where we have many cores that can either work on their own, or cooperatively to solve a problem.

It was thus found to be necessary to move from sequential programs that we all love to write to parallel programs. In parallel programs, the programmer needs to tell the compiler and hardware which parts of the code can run in parallel. For example, consider a loop with N iterations, where the iterations are independent of each other – they do not have data dependences. In this case, we run the code corresponding to each of the N iterations, on a different core. We can use N cores for this purpose. In theory we can speedup up the program by N times. However, in practice the number will be much lower. This is because often there will be some dependences between the iterations. These need to be managed by complex hardware and software mechanisms. Such actions slow down the program.

Multicore processors and parallel processing are very rich areas. People have been working on parallel programming models for at least the last three decades, and have also been designing hardware to ensure that parallel programs running on different cores run efficiently. Furthermore, if there is a need for them

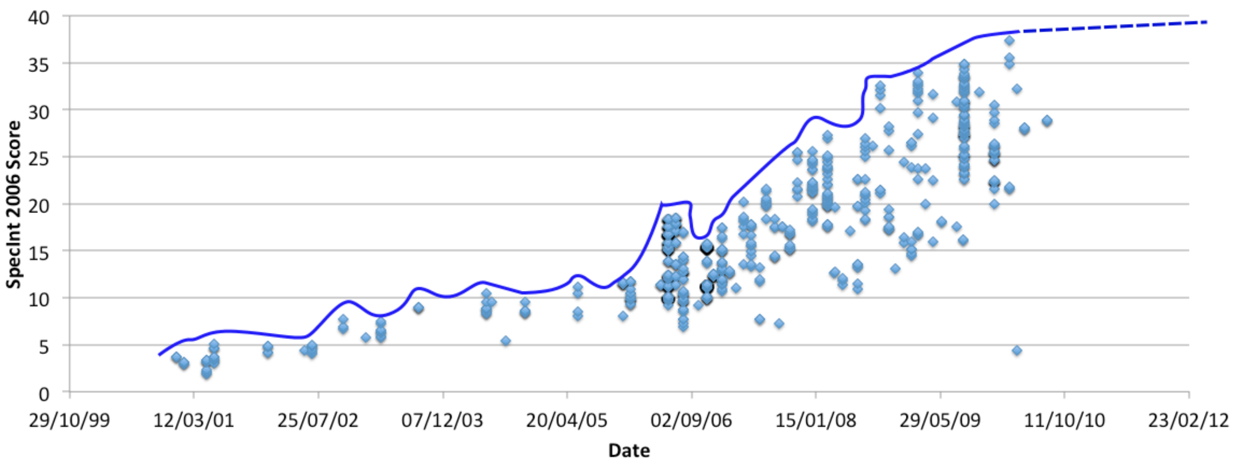

Figure 1.5: Improvement in processor performance over the years

to communicate with each other then the hardware needs to facilitate this with appropriate correctness guarantees.

1.2.1 GPUs

The first category of hardware that we describe in this space is the area of graphics processors (known as GPUs). GPUs were originally used for processing graphics and rendering complex scenes. Hence, GPUs have traditionally been used in games, CAD software, and software to create computer graphics videos. However, the last 10 years have seen a massive adoption of GPUs in conventional high performance computing such as numerical simulation, weather prediction, and finite element modelling of physical systems. GPUs are a very good fit for such applications because they consist of hundreds of very small cores – 512 to 4096 as of 2020. These cores are capable of performing arithmetic and logical calculations. If we have highly computation intensive numerical code without a lot of branches and irregular memory accesses, a GPU can be used. For example, if we want to compute the direction a cyclone will take, or its severity after three days, we need to perform a very large and complex weather simulation. Such programs are often based on large matrix operations, which have a lot of mutually independent computations, and this is where massively parallel GPUs can be used. A typical server processor can provide a maximum throughput of 10 GFLOPS (1 GFLOP = 10^9 floating point operations per second). In comparison the latest GPU can provide roughly 15 TFLOPS (1 TFLOP = 1000 GFLOPS) of computational throughput.

It should be, however, understood that GPUs are useful for a limited set of programs, particularly those that are based on large matrix operations. However, in general for parallel programs with complex communication patterns between different programs running on different cores, GPUs are not suitable. We shall delve into such issues in Chapter 6.

1.2.2 Large Multicore Processors

Since GPUs are not particularly very useful for complicated parallel programs, we need to create large multicore processors, where we have 10-20 cores on a single chip. These cores have complicated OOO pipelines. The OOO pipelines can support programs with complex branching and memory access patterns. However, for such processors to be effectively used, it is necessary to create hardware support for their communication model. Communication is typically done through reads and writes to the memory system. Hence, we need a fairly sophisticated memory system that can provide a set of intuitive correctness guarantees to parallel programs such that they behave more or less like equivalent sequential programs. Such mechanisms are known as cache coherence and memory consistency (see Chapter 9), and necessitate a lot of elaborate hardware support.

Transactional Memory

Parallel programming on multicore processors is regarded as difficult. As long as expert programmers in industry were writing such code, this was acceptable. However, to take parallel programming to the masses, it is necessary to have good programming models. These models need to ensure the correctness of programs, and also make it easy to write such programs. Furthermore, if there are some special programming mechanisms that are a part of these models, we might need additional hardware support.

Parallel programming models have been getting easier with time particularly with the parallel hardware becoming very cheap. The moment something needs to permeate to the masses, it needs to become very easy to use. Normally, programming languages have special functions and markers to mark regions as *parallel*.

However, to take things one step further, a new model that has been proposed, and is becoming increasingly popular is *transactional memory*. Here, we mark regions of code that need to be executed in parallel and should also appear to execute instantaneously. This is a very simple programming model, and even beginners can write a complex parallel program with little training. There are two kinds of transactional memory: software transactional memory (STM) and hardware transactional memory (HTM). STMs typically do not require any hardware support, whereas, HTMs require hardware support. The latter are far more efficient; however, they require complex hardware changes. We shall discuss such trade-offs in Chapter 9.

1.3 High Performance Memory System

To support high performance multi-core processors, we need a fast and high bandwidth memory system. The memory system needs to deliver both data and instructions to these cores such that they do not get stalled due to a lack of instructions or data. This requires us to design a large system of memories (known as *cache banks*), where we can quickly locate data, and supply it to the requesting cores. We shall see in Chapter 7 that there are many ways of very efficiently designing memories such that we can decrease their access times, and increase their bandwidth.

There are two kinds of memories that we typically have in modern architectures: on-chip caches based on SRAM (static RAM) cells, and off-chip memories made of DRAM (dynamic RAM) cells. We dedicate one chapter to study techniques to design efficient SRAM memories (Chapter 7), and one more chapter to study the design of DRAM memories (Chapter 10).

Just having a fast memory structure is not enough. It is important to route messages swiftly to these memory elements and get quick responses. We thus need an extremely responsive on-chip network that can provide low latency and high bandwidth. When the number of cores and on-chip memory elements was limited to single digits, the design of the on-chip network (referred to as the NoC, network-on-chip) was not very important. However, in modern processors with 20+ cores, and a similar number of memory elements, the NoC has suddenly become important.

Furthermore, we have processors such as the Intel® Xeon Phi™ processor with 70-80 cores. In such processors, the challenge of designing fast and power efficient interconnects is even more severe. Along with the traditional drivers – performance and power – we have several other issues with today's NoCs such as congestion, avoiding network traffic deadlocks, effective routing, and reliability. These issues will be covered in detail in Chapter 8.

1.4 Power, Temperature, Parameter Variation, and Reliability

Till the end of the late 90s, power was not considered an important design criteria. However, very soon power consumption became a first order design criteria. There were several reasons for this. The first was that increased power led to increased temperature. Unfortunately, leakage power (static power) is a super-linear function of the temperature. Thus, this leads to a positive feedback loop between power and temperature. Increased power dissipation leads to higher temperature, which in turn leads to increased leakage power, and so on. In extreme cases, this process might not converge and can lead to a condition called *thermal runaway* where the leakage power increases prohibitively. The results of this can be catastrophic. Additionally, there are other physical limitations: limits to conventional air cooling, and for embedded devices the battery capacity is limited. Hence, we need to design techniques to reduce power and energy consumption. This is discussed in Chapter 11.

To further compound the problem of high leakage power, we have the issue of *parameter variation* – variations in the fabrication process, supply voltage, and temperature. *Process variation* is the variation induced in the physical properties of transistors and interconnects due to the limitations in the physical processes used to create such devices. For example, to fabricate a 14 nm structure, we use light with a wavelength of 193 nm. This leads to non-uniformity in the design, and often when we are trying to print a rectangle on silicon, we actually fabricate a rounded rectangle with deformed edges. This leads to two pernicious effects: high leakage power and low performance. It is necessary to design fabrication methods and computer architectures to effectively combat this problem, and design hardware that is relatively immune to the negative effects of process variation. In addition, there can be fluctuations in the temperature of different regions of a chip due to a varying amount of power consumption. High temperature makes circuits dissipate more leakage power. We also can have fluctuations in the supply voltage because of resistive and inductive drops in the chip's power grid. If the supply voltage reduces, the transistors become slower. If we consider all of these effects, designing a high performance processor becomes fairly difficult unless we provide large margins for performance and power. We shall look at state of the art solutions in Chapter 12.

Other than increasing leakage power, there are many more reasons why high chip temperature is considered to be absolutely undesirable. High temperature increases the rate of failure of wires and transistors exponentially. If we have high temperature for a sustained period, then we can have permanent failures where for example a wire gets snapped, or a transistor gets destroyed. Furthermore, as we shall see in Chapter 12 high temperature can also induce transistor ageing, where its properties gradually degrade with time. Hence, there is now a large body of research dedicated to reducing power and temperature. Note that it is not necessary that a power control algorithm will always reduce the peak temperature. Here, the "peak temperature" is important because this is where we will have the highest rate of transistor and interconnect ageing. Similarly, temperature control algorithms need not be the best schemes to keep power consumption in check.

Other than faults caused by high temperature, there are sadly many other mechanisms that can cause a chip to malfunction either temporarily or permanently. One such cause is bit flips due to the impact of alpha particles or cosmic rays. These are known as *soft errors*. A soft error is typically a very short and transient phenomenon where an impact with a charged ion leads to a current pulse, which can flip the value of a bit that is either being computed by an ALU or stored in a latch. There are many ways to make circuits relatively more immune to soft errors. We shall look at such methods in Chapter 12 along with some other lesser known methods of failure.

1.5 Security

Nowadays, no processor design effort or course on computer architecture is complete without discussing the issue of security – both software and hardware. Security is gradually becoming more important because processors are beginning to get embedded everywhere. In the 70s computers were confined to either large companies or universities. However, today everybody has several computers with her: a phone, a tablet, and a smart watch. Small processors are embedded in smart glasses, and medical devices such as pacemakers. Performance and power efficiency are nevertheless still very worthy goals to pursue. However, we also need to ensure that these devices run correctly in the field. Even after taking all the steps to guarantee the reliability of the device, we need to ensure that malicious users cannot subvert its security measures, and either cause the device to malfunction, or steal sensitive data belonging to other users. Imagine a hacker breaking into a pacemaker. The hacker can cause it to fail, thus endangering the life of the patient.

Hence, it is very important to ensure that both code and data are secure. We shall discuss different methods to secure code and data in both the processor and the memory in Chapter 13.

1.6 Architectures for Machine Learning

General purpose computing is approaching its limits. The feature sizes have already reached 7 nm. They are not expected to go below 5 nm. Subsequently, traditional silicon based architectures will saturate because faster and more power efficient transistors may not be available unless there is a significant shift in the technology. Till now this was not a problem. Architects were always able to leverage improvements in process technology to design more efficient processor designs. For example, with reducing transistor sizes architects were able to place larger L2 caches on chip. They could then further propose many techniques to make caches and the interconnecting network more efficient. But with this process expected to come to a stop, the next frontier is to build application-specific processors. One of the most promising applications is machine learning (ML) accelerators. There are also many allied fields that use ML techniques such as computer vision, speech processing, and data analytics. All of them will benefit with such application-specific ML accelerators.

We shall discuss different architectures for machine learning in Chapter 14. This field has significantly grown and matured over the last 5 years. We now have numerous architectures for a host of machine learning algorithms. The most prominent class of architectures try to optimise the execution of CNNs (convolutional neural networks). CNNs have by far seen the maximum number of applications in the field of image, speech, and data processing. These techniques are generic in nature and can also be used to create a very large number of other ML architectures as well.