Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/cor

Computer Architecture

1School of Computer, Wuhan University, Wuhan 430072, China
2Department of computer science, Huazhong Normal University, Wuhan 430079, spain
(2013)

Abstract
Computer architecture is often used to refer to an area of specialization within the academic discipline of computer science. At other times, the phrase computer architecture may be used to refer to a specific machine specification, such as SPARC, Intel X86, PowerPC, or Motorola 680x0. A more precise designation for specifying such machines is instruction set architecture (ISA). In the 1980's, when computer architecture programs first began within computer science departments at various universities, heavy emphasis was placed on research related to designing better ISA's. Nowadays, the industry has converged on a few different ISA's and tends to rebuild them over and over again with faster hardware, so computer architecture has moved to focus more on the ways to speed up computer hardware and system-level software than on designing new ISA's. The academic discipline of computer architecture thus tends to overlap somewhat with fields called computer engineering, or the like, that usually are located within the academic discipline of electrical engineering. The field known as computer architecture may touch all aspects of how specific computers can be specified, built, and verified (tested).

Computer architecture comprises at least three main subcategories:

- Instruction set architecture, or ISA, is the abstract image of a computing system that is seen by a machine language (or assembly language) programmer, including the instruction set, word size, memory address modes, processor registers, and address and data formats.
- Micro architecture, also known as Computer organization is a lower level, more concrete and detailed, description of the system that involves how the constituent parts of the system are interconnected and how they interoperate in order to implement the ISA.[2] The size of a computer's cache for instance, is an organizational issue that generally has nothing to do with the ISA.
- System Design which includes all of the other hardware components within a computing system such as:

  1. System interconnects such as computer buses and switches
  2. Memory controllers and hierarchies
  3. CPU off-load mechanisms such as direct memory access (DMA)
  4. Issues like multiprocessing.

Once both ISA and micro architecture have been specified, the actual device needs to be designed into hardware. This design process is called the implementation. Implementation is usually not considered architectural definition, but rather hardware design engineering.

Implementation can be further broken down into three (not fully distinct) pieces:

- Logic Implementation — design of blocks defined in the micro architecture at (primarily) the register-transfer and gate levels.

- Circuit Implementation — transistor-level design of basic elements (gates, multiplexers, latches etc.) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at this level, or even (partly) at the physical level, for performance reasons.
- Physical Implementation — physical circuits are drawn out, the different circuit components are placed in a chip floor plan or on a board and the wires connecting them are routed.

For CPUs, the entire implementation process is often called CPU design.

More specific usages of the term include more general wider-scale hardware architectures, such as cluster computing and Non-Uniform Memory Access (NUMA) architectures.

There are many different kinds of computer architectures. one way of categorizing computer architectures is by number of instructions executed per clock. Many computing machines read one instruction at a time and execute it (or they put a lot of effort into acting as if they do that, even if internally they do fancy superscalar and out-of-order stuff). I call such machines "von Neumann" machines, because all of them have a von Neumann bottleneck. Such machines include CISC, RISC, MISC, TTA, and DSP architectures. Such machines include accumulator machines, register machines, and stack machines. Other machines read and execute several instructions at a time (VLIW, super-scalar), which break the one-instruction-per-clock limit, but still hit the von Neumann bottleneck at some slightly larger number of instructions-per-clock. Yet other machines are not limited by the von Neumann bottleneck, because they pre-load all their operations once at power-up and then process data with no further instructions. Such non-Von-Neumann machines include dataflow architectures, such as systolic architectures and cellular automata, often implemented with FPGAs, and the NON-VON supercomputer.

Another way of categorizing computer architectures is by the connection(s) between the CPU and memory. Some machines have a unified memory, such that a single address corresponds to a single place in memory, and when that memory is RAM, one can use that address to read and write data, or load that address into the program counter to execute code. I call these machines Princeton machines. Other machines have several separate memory spaces, such that the program counter always refers to "program memory" no matter what address is loaded into it, and normal reads and writes always go to "data memory", which is a separate location usually containing different information even when the bits of the data address happen to be identical to the bits of the program memory address. Those machines are "pure Harvard" or "modified Harvard" machines. Most DSPs have 3 separate memory areas -- the X ram, the Y ram, and the program memory. The DSP, Princeton, and 2-memory Harvard machines are three different kinds of von Neumann machines. A few machines take advantage of the extremely wide connection between memory and computation that is possible when they are both on the same chip -- computational ram or iRAM or CAM RAM -- which can be seen as a kind of non-von Neumann machine.

A few people use a narrow definition of "von Neumann machine" that does not include Harvard machines. If you are one of those people, then what term would you use for the more general concept of "a machine that has a von Neumann bottleneck", which includes both Harvard and Princeton machines, and excludes NON-VON?

Most embedded systems use Harvard architecture. A few CPUs are "pure Harvard", which is perhaps the simplest arrangement to build in hardware: the address bus to the read-only program memory is exclusively is connected to the program counter, such as many early Microchip PICmicros. Some modified Harvard machines, in addition, also put constants in program memory, which can be read with a special "read constant data from program memory" instruction (different from the "read from data memory" instruction). The software running in the above kinds of Harvard machines cannot change the program memory, which is effectively ROM to that software. Some embedded systems are "self-programmable", typically with program memory in flash memory and a special "erase block of flash memory" instruction and a special "write block of flash memory" instruction (different from the normal "write to data memory" instruction), in addition to the "read data from program memory" instruction. Several more recent Microchip PICmicros and Atmel AVRs are self-programmable modified Harvard machines.

Another way to categorize CPUs is by their clock. Most computers are synchronous -- they have a single global clock. A few CPUs are asynchronous -- they don't have a clock -- including the ILLIAC I and ILLIAC II, which at one time were the fastest supercomputers on earth.

There are many types of computer architectures:

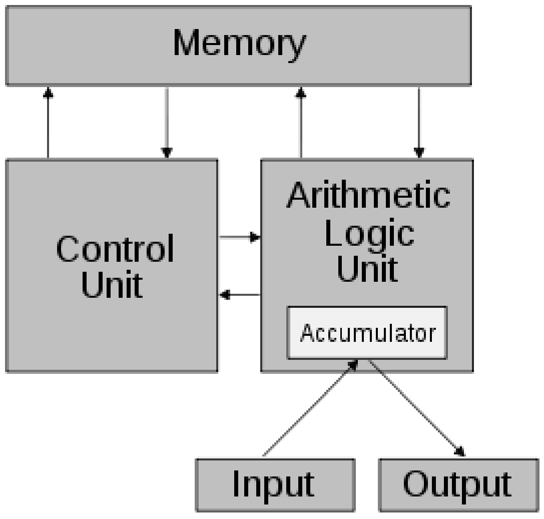
- Quantum computer vs. Chemical computer
- Scalar processor vs. Vector processor
- Non-Uniform Memory Access (NUMA) computers
- Register machine vs. Stack machine
- Harvard architecture vs. von Neumann architecture
- Cellular architecture

In the following we will focus on some major architectures

- Harvard architecture vs. von Neumann architecture
- Quantum computer vs. Chemical computer
- Cellular architecture

Von Neumann Architecture

The von Neumann architecture is a design model for a stored-program digital computer that uses a central processing unit (CPU) and a single separate storage structure ("memory") to hold both instructions and data. It is named after the mathematician and early computer scientist John von Neumann. Such computers implement a universal Turing machine and have a sequential architecture.



Schematic of the von Neumann architecture. The Control Unit and Arithmetic Logic Unit form the main components of the Central Processing Unit (CPU).

A stored-program digital computer is one that keeps its programmed instructions, as well as its data, in read-write, random-access memory (RAM). Stored-program computers were advancement over the program-controlled computers of the 1940s, such as the Colossus and the ENIAC, which were programmed by setting switches and inserting patch leads to route data and to control signals between various functional units. In the vast majority of modern computers, the same memory is used for both data and program instructions. The mechanisms for transferring the data and instructions between the CPU and memory are, however, considerably more complex than the original von Neumann architecture.

The terms "von Neumann architecture" and "stored-program computer" are generally used interchangeably, and that usage is followed in this article.

Description

The earliest computing machines had fixed programs. Some very simple computers still use this design, either for simplicity or training purposes. For example, a desk calculator (in principle) is a fixed program computer. It can do basic mathematics, but it cannot be used as a word processor or a gaming console. Changing the program of a fixed-program machine requires re-wiring, re-structuring, or re-designing the machine. The earliest computers were not so much "programmed" as they were "designed". "Reprogramming", when it was possible at all, was a laborious process, starting with flowcharts and paper notes, followed by detailed engineering designs, and then the often-arduous process of physically re-wiring and re-building the machine. It could take three weeks to set up a program on ENIAC and get it working.

The idea of the stored-program computer changed all that: a computer that by design includes an instruction set and can store in memory a set of instructions (a program) that details the computation.

A stored-program design also lets programs modify themselves while running. One early motivation for such a facility was the need for a program to increment or otherwise modify the address portion of instructions, which had to be done manually in early designs. This became less important when index registers and indirect addressing became usual features of machine architecture. Self-modifying code has largely fallen out of favor, since it is usually hard to understand and debug, as well as being inefficient under modern processor pipelining and caching schemes.

On a large scale, the ability to treat instructions as data is what makes assemblers, compilers and other automated programming tools possible. One can "write programs which write programs". On a smaller scale, I/O-intensive machine instructions such as the BITBLT primitive used to modify images on a bitmap display, were once thought to be impossible to implement without custom hardware. It was shown later that these instructions could be implemented efficiently by "on the fly compilation" ("just-in-time compilation") technology, e.g., code-generating programs—one form of self-modifying code that has remained popular.

There are drawbacks to the von Neumann design. Aside from the von Neumann bottleneck described below, program modifications can be quite harmful, either by accident or design. In some simple stored-program computer designs, a malfunctioning program can damage itself, other programs, or the operating system, possibly leading to a computer crash. Memory protection and other forms of access control can usually protect against both accidental and malicious program modification.

Von Neumann bottleneck

The separation between the CPU and memory leads to the von Neumann bottleneck, the limited throughput (data transfer rate) between the CPU and memory compared to the amount of memory. In most modern computers, throughput is much smaller than the rate at which the CPU can work. This seriously limits the effective processing speed when the CPU is required to perform minimal processing on large amounts of data. The CPU is continuously forced to wait for needed data to be transferred to or from memory. Since CPU speed and memory size have increased much faster than the throughput between them, the bottleneck has become more of a problem, a problem whose severity increases with every newer generation of CPU.

The term "von Neumann bottleneck" was coined by John Backus in his 1977 ACM Turing Award lecture. According to Backus:

> *Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning*

and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

The performance problem can be alleviated (to some extent) by several mechanisms. Providing a cache between the CPU and the main memory, providing separate caches with separate access paths for data and instructions (the so-called Harvard architecture), and using branch predictor algorithms and logic are three of the ways performance is increased. The problem can also be sidestepped somewhat by using parallel computing, using for example the NUMA architecture—this approach is commonly employed by supercomputers. It is less clear whether the intellectual bottleneck that Backus criticized has changed much since 1977. Backus's proposed solution has not had a major influence.[citation needed] Modern functional programming and object-oriented programming are much less geared towards "pushing vast numbers of words back and forth" than earlier languages like Fortran were, but internally, that is still what computers spend much of their time doing, even highly parallel supercomputers.

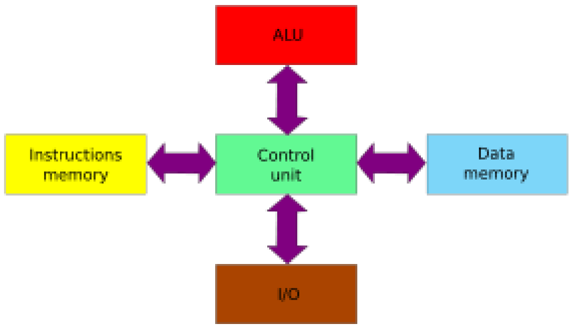
Early von Neumann-architecture computers

The First Draft described a design that was used by many universities and corporations to construct their computers. Among these various computers, only ILLIAC and ORDVAC had compatible instruction sets.

- ORDVAC (U-Illinois) at Aberdeen Proving Ground, Maryland (completed Nov 1951)
- IAS machine at Princeton University (Jan 1952)
- MANIAC I at Los Alamos Scientific Laboratory (Mar 1952)
- ILLIAC at the University of Illinois, (Sept 1952)
- AVIDAC at Argonne National Laboratory (1953)
- ORACLE at Oak Ridge National Laboratory (Jun 1953)
- JOHNNIAC at RAND Corporation (Jan 1954)
- BESK in Stockholm (1953)
- BESM-1 in Moscow (1952)
- DASK in Denmark (1955)
- PERM in Munich (1956)
- SILLIAC in Sydney (1956)
- WEIZAC in Rehovoth (1955)

Harvard Architecture

The Harvard architecture is computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters. These early machines had limited data storage, entirely contained within the central processing unit, and provided no access to the instruction storage as data. Programs needed to be loaded by an operator, the processor could not boot itself.



Harvard architecture

Today, most processors implement such separate signal pathways for performance reasons but actually implement Modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it.

In Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

Contrast with von Neumann architectures

Under pure von Neumann architecture the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache. A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight bit byte that isn't part of that twenty-four bit value.

Contrast with Modified Harvard architecture

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common address space. While the CPU executes from cache, it acts as a pure Harvard machine. When accessing backing memory, it acts like a von Neumann machine (where code can be moved around like data, a powerful technique). This modification is widespread in modern processors such as the ARM architecture and X86 processors. It is sometimes loosely called Harvard architecture, overlooking the fact that it is actually "modified".

Another modification provides a pathway between the instruction memory (such as ROM or flash) and the CPU to allow words from the instruction memory to be treated as read-only data. This technique is used in some microcontrollers, including the Atmel AVR. This allows constant data, such as text strings or function tables, to be accessed without first having to be copied into data memory, preserving scarce (and power-hungry) data memory for read/write variables. Special machine language instructions are provided to read data from the instruction memory. (This is distinct from instructions which themselves embed constant data, although for individual constants the two mechanisms can substitute for each other.)

Speed

In recent years, the speed of the CPU has grown many times in comparison to the access speed of the main memory. Care needs to be taken to reduce the number of times main memory is accessed in order to maintain performance. If, for instance, every instruction run in the CPU requires an access to memory, the computer gains nothing for increased CPU speed—a problem referred to as being "memory bound".

It is possible to make extremely fast memory but this is only practical for small amounts of memory for cost, power and signal routing reasons. The solution is to provide a small amount of very fast memory known as a CPU cache which holds recently accessed data. As long as the memory that the CPU needs is in the cache, the performance hit is much smaller than it is when the cache has to turn around and get the data from the main memory.

Internal vs. external design

Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. In particular, the Modified Harvard architecture is very common. CPU cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not formally divided into separate instruction and data sections, although it may well have separate memory controllers used for concurrent access to RAM, ROM and (NOR) flash memory.

Thus, while a von Neumann architecture is visible in some contexts, such as when data and code come through the same memory controller, the hardware implementation gains the efficiencies of the Harvard architecture for cache accesses and at least some main memory accesses.

In addition, CPUs often have write buffers which let CPUs proceed after writes to non-cached regions. The von Neumann nature of memory is then visible when instructions are written as data by the CPU and software must ensure that the caches (data and instruction) and write buffer are synchronized before trying to execute those just-written instructions.

Quantum Computer

A quantum computer is a device for computation that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform operations on data. Quantum computers are different from traditional computers based on transistors. The basic principle behind quantum computation is that quantum properties can be used to represent data and perform operations on these data. A theoretical model is the quantum Turing machine, also known as the universal quantum computer.

Although quantum computing is still in its infancy, experiments have been carried out in which quantum computational operations were executed on a very small number of qubits (quantum bit). Both practical and theoretical research continues, and many national government and military funding agencies support quantum computing research to develop quantum computers for both civilian and national security purposes, such as cryptanalysis.

If large-scale quantum computers can be built, they will be able to solve certain problems much faster than any current classical computers (for example Shor's algorithm). Quantum computers do not allow the computation of functions that are not theoretically computable by classical computers, i.e. they do not alter the Church–Turing thesis. The gain is only in efficiency.

Basis

A classical computer has a memory made up of bits, where each bit represents either a one or a zero. A quantum computer maintains a sequence of qubits. A single qubit can represent a one, a zero, or, crucially, any quantum superposition of these; moreover, a pair of qubits can be in any quantum superposition of 4 states, and three qubits in any superposition of 8. In general a quantum computer with n qubits can be in an arbitrary superposition of up to 2n different states simultaneously (this compares to a normal computer that can only be in one of these 2n states at any one time). A quantum computer operates by manipulating those qubits with a fixed sequence of quantum logic gates. The sequence of gates to be applied is called a quantum algorithm.

An example of an implementation of qubits for a quantum computer could start with the use of particles with two spin states: "down" and "up" (typically written $|\downarrow\rangle$ and $|\uparrow\rangle$, or $|0\rangle$ and $|1\rangle$). But in fact any system possessing an observable quantity A which is conserved under time evolution and such that A has at least two discrete and sufficiently spaced consecutive eigenvalues, is a suitable candidate for implementing a qubit. This is true because any such system can be mapped onto an effective spin-1/2 system.

Operation

While a classical three-bit state and a quantum three-qubit state are both eight-dimensional vectors, they are manipulated quite differently for classical or quantum computation. For computing in either case, the system must be initialized, for example into the all-zeros string, $|000\rangle$, corresponding to the vector (1,0,0,0,0,0,0,0). In classical randomized computation, the system evolves according to the application of stochastic matrices, which preserve that the probabilities add up to one (i.e., preserve the L1 norm). In quantum computation, on the other hand, allowed operations are unitary matrices, which are effectively rotations (they preserve that the sum of the squares adds up to one, the Euclidean or L2 norm). (Exactly what unitaries can be applied depend on the physics of the quantum device.) Consequently, since rotations can be undone by rotating backward, quantum computations are reversible. (Technically, quantum operations can be probabilistic combinations of unitaries, so quantum computation really does generalize classical computation. See quantum circuit for a more precise formulation.)

Finally, upon termination of the algorithm, the result needs to be read off. In the case of a classical computer, we sample from the probability distribution on the three-bit register to obtain one definite three-bit string, say 000. Quantum mechanically, we measure the three-qubit state, which is equivalent to collapsing the quantum state down to a classical distribution (with the coefficients in the classical state being the squared magnitudes of the coefficients for the quantum state, as described above) followed by sampling from that distribution. Note that this destroys the original quantum state. Many algorithms will only give the correct answer with a certain probability, however by repeatedly initializing, running and measuring the quantum computer, the probability of getting the correct answer can be increased.

Chemical Computer

A chemical computer, also called reaction-diffusion computer, BZ computer or gooware computer is an unconventional computer based on a semi-solid chemical "soup" where data is represented by varying concentrations of chemicals. The computations are performed by naturally occurring chemical reactions. So far it is still in a very early experimental stage, but may have great potential for the computer industry.

Rationale

The simplicity of this technology is one of the main reasons why it in the future could turn into a serious competitor to machines based on conventional hardware. A modern microprocessor is an incredibly complicated device that can be destroyed during production by no more than a single airborne microscopic particle. In contrast a cup of chemicals is a simple and stable component that is cheap to produce.

In a conventional microprocessor the bits behave much like cars in city traffic; they can only use certain roads, they have to slow down and wait for each other in crossing traffic, and only one driving field at once can be used. In a BZ solution the waves are moving in all thinkable directions in all dimensions, across, away and against each other. These properties might make a chemical computer able to handle billions of times more data than a traditional computer. An analogy would be the brain; even if a microprocessor can transfer information much faster than a neuron, the brain is still much more effective for some tasks because it can work with a much higher amount of data at the same time.

Basic Principles

The wave properties of the BZ reaction means it can move information in the same way as all other waves. This still leaves the need for computation, performed by conventional microchips using the binary code transmitting and changing ones and zeros through a complicated system of logic gates. To perform any conceivable computation it is sufficient to have NAND gates. (A NAND gate has two bits input. Its output is 0 if both bits are 1, otherwise it's 1). In the chemical computer version logic gates are implemented by concentration waves blocking or amplifying each other in different ways.

Cellular Architecture

A cellular architecture is a type of computer architecture prominent in parallel computing. Cellular architectures are relatively new, with IBM's Cell microprocessor being the first one to reach the market. Cellular architecture takes multi-core architecture design to its logical conclusion, by giving the programmer the ability to run large numbers of concurrent threads within a single processor. Each 'cell' is a compute node containing thread units, memory, and communication. Speed-up is achieved by exploiting thread-level parallelism inherent in many applications.

Cell, a cellular architecture containing 9 cores, is the processor used in the PlayStation 3. Another prominent cellular architecture is Cyclops64, a massively parallel architecture currently under development by IBM.

Cellular architectures follow the concrete programming paradigm, which exposes the programmer to much of the underlying hardware. This allows the programmer to greatly optimize his code for the platform, but at the same time makes it more difficult to develop software.

Cellular architecture builds next generation supercomputers

Early in 2000, a group of IBM researchers and engineers began to solve this problem using a future generation of ultra-high performance computers. The group proposed a new computer design, called cellular architecture. "The key idea is that instead of focusing on processor micro-architecture and structure, as in the past, we optimize the memory system's latency and throughput—how fast we can access, search and move data," explains Dean.

Cellular computers, like existing supercomputers, consist of many identical processors. However, cellular computers differ from supercomputers in a number of important ways, each designed to speed memory access. "A cellular machine has memory units integrated into each of its cells, instead of having a central bank accessed by all the processors," points out George Chiu, one of the scientists working on IBM's effort. This means that processors don't have to take turns waiting for access to central memory. In fact, in one of IBM's experimental cellular machines, Blue Gene, each processor will be fed data by eight different memory sub-sectors, each connected with a different subtask or thread. "It's sort of like having a chicken run from one feed spigot to another so the chicken is always feeding," Chiu says.

Second, when data is needed, all cells can be used to look for it in their own memories, instead of having a central controller deciding which processors should do the job. "This means that some processors will waste time looking for data that is not in their cells' memory, but the data will be found more quickly this way," says Dean.

For this architecture to work on very large databases there must be a high number of cells, with tens of thousands to a million processors, rather than the hundreds of processors found in existing supercomputers. Also, unlike existing supercomputers, each cell is small enough—a single chip—to enable this extremely large-scale parallel operation (the ability to divvy up instructions and perform them simultaneously). To shorten communication time between cells, each is connected only to its nearest neighbors.

Finally, with so many processors, some failures are inevitable, so cellular architecture automatically reroutes instructions and data around malfunctioning cells.

"Cellular architecture uses ideas that have already been developed, such as putting a whole system—memory and processors—on a chip, and using many processors in parallel to carry out instructions, but pushes them to extremes to get the fastest data acquisition and search possible," says Dean.

References

[1] J. Campbell, “Speaker recognition: a tutorial,” *Proc. IEEE*, vol. 85, pp. 1437–1462, Sept. 1997.

[2] D. A. Reynolds, T. Quatieri, and R. Dunn, “Speaker verification using adapted Gaussian mixture models,” *Digital Signal Processing*, vol. 10, no. 1–3, pp. 19–41, 2000.

[3] D. A. Reynolds, “Comparison of background normalization methods for text-independent speaker verification,” in *Proc. Eurospeech*, 1997.

[4] D. A. Reynolds and R. C. Rose, “Robust text-independent speaker identification using Gaussian mixture speaker models,” *IEEE Trans. Speech Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995.

[5] J. L. Gauvain and C.-H. Lee, “Maximum a posteriori estimation for multivariate Gaussian mixture observations of Markov chains,” *IEEE Trans. Speech Audio Processing*, vol. 2, pp. 291–298, Apr. 1994.

[6] E. Bocchieri, “Vector quantization for the efficient computation of continuous density likelihoods,” in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1993, pp. 692–695.

[7] K. M. Knill, M. J. F. Gales, and S. J. Young, “Use of Gaussian selection in large vocabulary continuous speech recognition using HMMs,” in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[8] D. B. Paul, “An investigation of Gaussian shortlists,” in *Proc. Automatic Speech Recognition and Understanding Workshop*, 1999.

[9] T. Watanabe, K. Shinoda, K. Takagi, and K.-I. Iso, “High speed speech recognition using tree-structured probability density function,” in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1995.

[10] J. Simonin, L. Delphin-Poulat, and G. Damnati, “Gaussian density tree structure in a multi-Gaussian HMM-based speech recognition system,” in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[11] T. J. Hanzen and A. K. Halberstadt, “Using aggregation to improve the performance of mixture Gaussian acoustic models,” in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1998.

[12] M. Padmanabhan, L. R. ahl, and D. Nahamoo, “Partitioning the feature space of a classifier with linear hyperplanes,” *IEEE Trans. Speech Audio Processing*, vol. 7, no. 3, pp. 282–288, 1999.

[13] R. Auckenthaler and J. Mason, “Gaussian selection applied to text-independent speaker verification,” in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[14] J. McLaughlin, D. Reynolds, and T. Gleason, “A study of computation speed-ups of the GMM-UBM speaker recognition system,” in *Proc. Eurospeech*, 1999.

[15] S. van Vuuren and H. Hermansky, “On the importance of components of the modulation spectrum of speaker verification,” in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[16] B. L. Pellom and J. H. L. Hansen, “An efficient scoring algorithm for Gaussian mixture model based speaker identification,” *IEEE Signal Processing Lett.*, vol. 5, no. 11, pp. 281–284, 1998.

[17] J. Oglesby and J. S. Mason, “Optimization of neural models for speaker identification,” in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1990, pp. 261–264.

[18] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe, “Global optimization of a neural network—hidden Markov model hybrid,” *IEEE Trans. Neural Networks*, vol. 3, no. 2, pp. 252–259, 1992.

[19] H. Bourlard and C. J. Wellekins, “Links between Markov models and multilayer perceptrons,” *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 12, pp. 1167–1178, Dec. 1990.

[20] J. Navrátil, U. V. Chaudhari, and G. N. Ramaswamy, “Speaker verification using target and background dependent linear transforms and multi-system fusion,” in *Proc. Eurospeech*, 2001.

[21] L. P. Heck, Y. Konig, M. K. Sonmez, and M. Weintraub, “Robustness to telephone handset distortion in speaker recognition by discriminative feature design,” *Speech Commun.*, vol. 31, pp. 181–192, 2000.

[22] A. Dempster, N. Laird, and D. Rubin, “Maximum likelihood from incomplete data via the EM algorithm,” *J. R. Statist. Soc.*, vol. 39, pp. 1–38, 1977.

[23] K. Shinoda and C. H. Lee, “A structural Bayes approach to speaker adaptation,” *IEEE Trans. Speech Audio Processing*, vol. 9, no. 3, pp. 276–287, 2001.

[24] K. Fukunaga, *Introduction to Statistical Pattern Recognition*. New York: Academic, 1990.

[25] J. C. Junqua, *Robust Speech Recogntion in Embedded Systems and PC*

[26] U. V. Chaudhari, J. Navrátil, S. H. Maes, and R. A. Gopinath, “Transformation enhanced multi-grained modeling for text-independent speaker recognition,” in *Proc. Int. Conf. Spoken Language Processing*, 2000.

[27] Q. Lin, E.-E. Jan, C. W. Che, D.-S. Yuk, and J. Flanagan, “Selective use of the speech spectrum and a VQGMM method for speaker identification,” in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[28] S. Raudys, *Statistical and Neural Classifiers: An Integrated Approach to Design*. New York: Springer, 2001.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986, pp. 318–364.

[30] [Online] Available: http://www.nist.gov/speech/tests/spk/index.htm.

[31] J. Pelecanos and S. Sridharan, “Feature warping for robust speaker verification,” in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[32] B. Xiang, U. V. Chaudhari, J. Navrátil, N. Ramaswamy, and R. A. Gopinath, “Short-time Gaussianization for robust speaker verification,” in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 2002.

[33] G. R. Doddington, M. A. Przybocki, A. F. Martin, and D. A. Reynolds, “The NIST speaker recognition evaluation—overview, methodology, systems, results, perspective,” *Speech Communication*, vol. 31, pp. 225–254, 2000.

Bing Xiang (M’03) was born in 1973 in China. He received the B.S. degree in radio and electronics and M.E. degree in signal and information processing from Peking University in 1995 and 1998, respectively. In January, 2003, he received the Ph.D. degree in electrical engineering from Cornell University, Ithaca, NY.

From 1995 to 1998, he worked on speaker recognition and auditory modeling in National Laboratory on Machine Perception, Peking University. Then he entered Cornell University and worked on speaker recognition and speech recognition in DISCOVER Lab as a Research Assistant. He also worked in the Human Language Technology Department of IBM Thomas J. Watson Research Center as a summer intern in both 2000 and 2001. He was a selected remote member of the SuperSID Group in the 2002 Johns Hopkins CLSP summer workshop in which he worked on speaker verification with high-lelvel information. In January, 2003, he joined the Speech and Language Processing Department of BBN Technologies where he is presently a Senior Staff Consultant-Technology. His research interests include large vocabulary speech recognition, speaker recognition, speech synthesis, keyword spotting, neural networks and statistical pattern recognition.

Toby Berger (S’60–M’66–SM’74–F’78) was born in New York, NY, on September 4, 1940. He received the B.E. degree in electrical engineering from Yale University, New Haven, CT in 1962, and the M.S. and Ph.D. degrees in applied mathematics from Harvard University, Cambridge, MA in 1964 and 1966, respectively.

From 1962 to 1968 he was a Senior Scientist at Raytheon Company, Wayland, MA, specializing in communication theory, information theory, and coherent signal processing. In 1968 he joined the faculty of Cornell University, Ithaca, NY where he is presently the Irwin and Joan Jacobs Professor of Engineering. His research interests include information theory, random fields, communication networks, wireless communications, video compression, voice and signature compression and verification, neuroinformation theory, quantum information theory, and coherent signal processing. He is the author/co-author of Rate Distortion Theory: A Mathematical Basis for Data Compression, Digital Compression for Multimedia: Principles and Standards, and Information Measures for Discrete Random Fields.

Dr. Berger has served as editor-in-chief of the IEEE Transactions on Information Theory and as president of the IEEE Information Theory