The future of Computer Science

Dave Cliff, University of Bristol

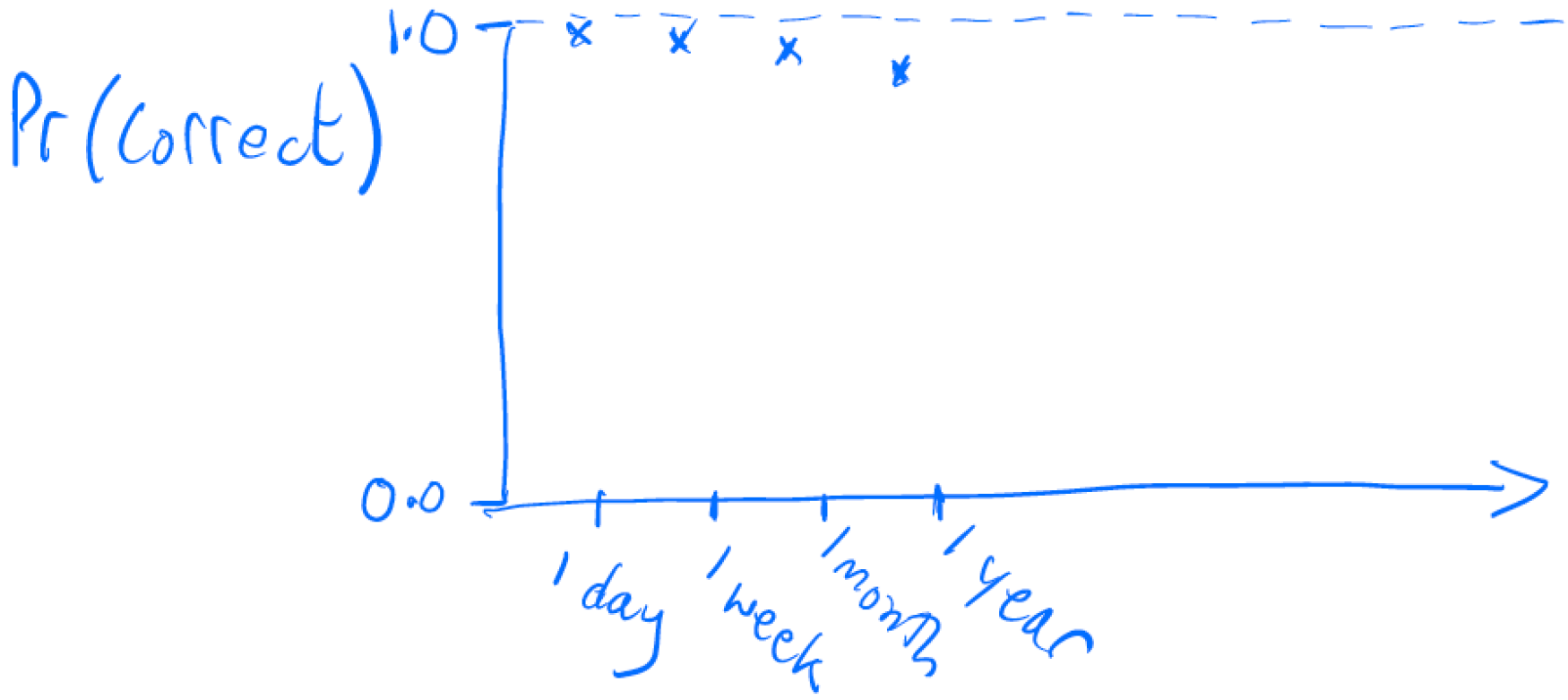
(and proud Leeds Class of ’87 graduate)

The partial future of Computer Science

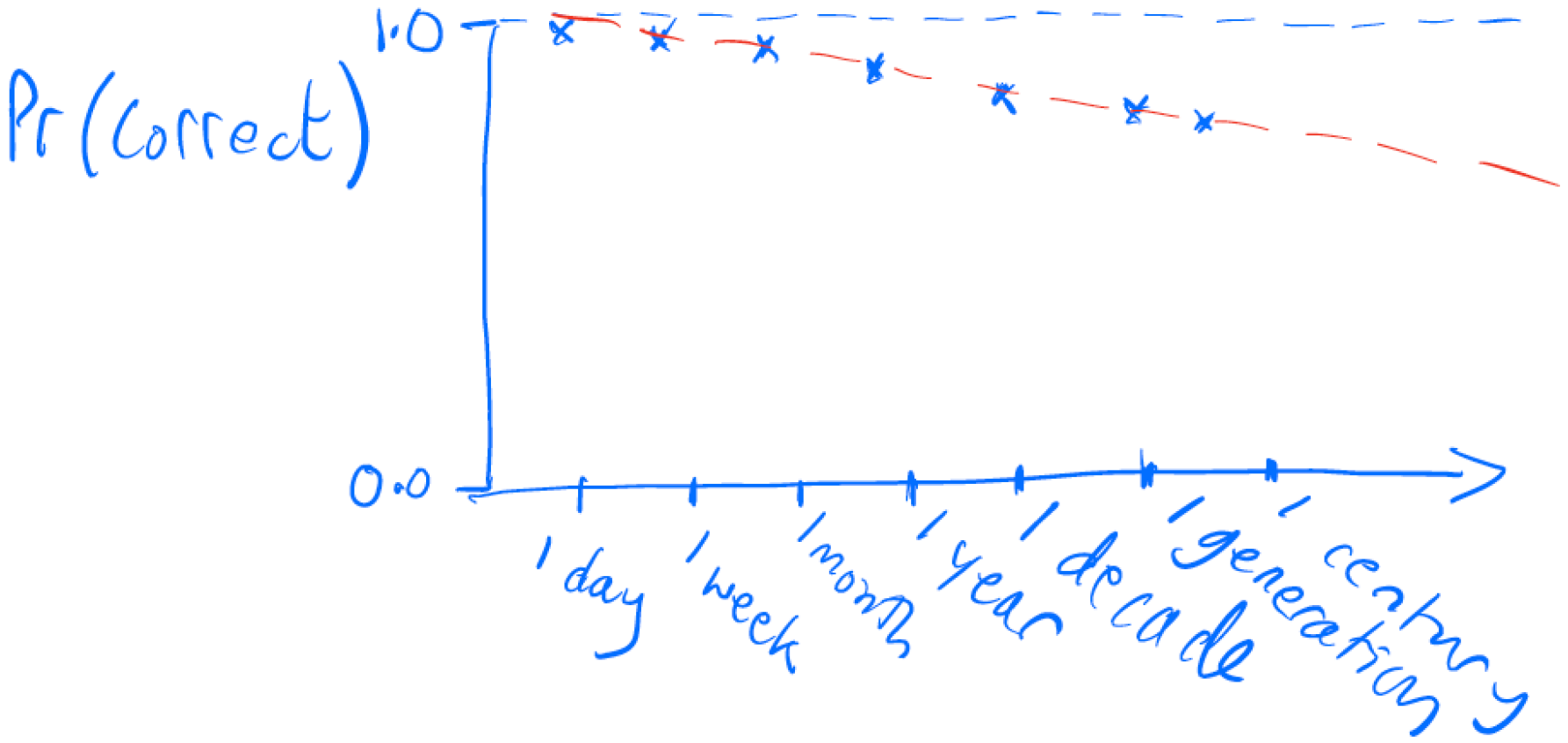
Dave Cliff, University of Bristol

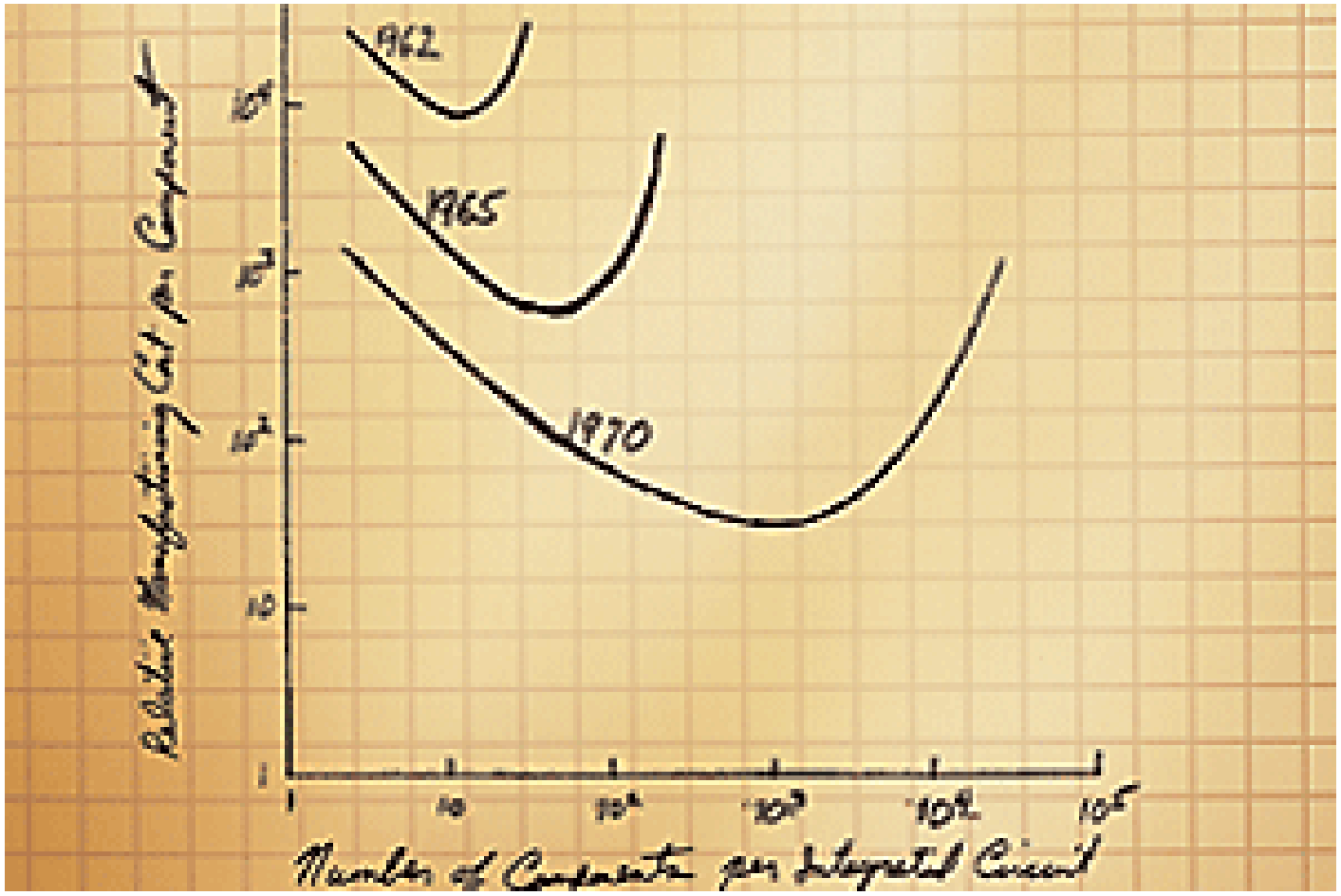
(and proud Leeds Class of ’87 graduate)

Predicting the future: near term easy

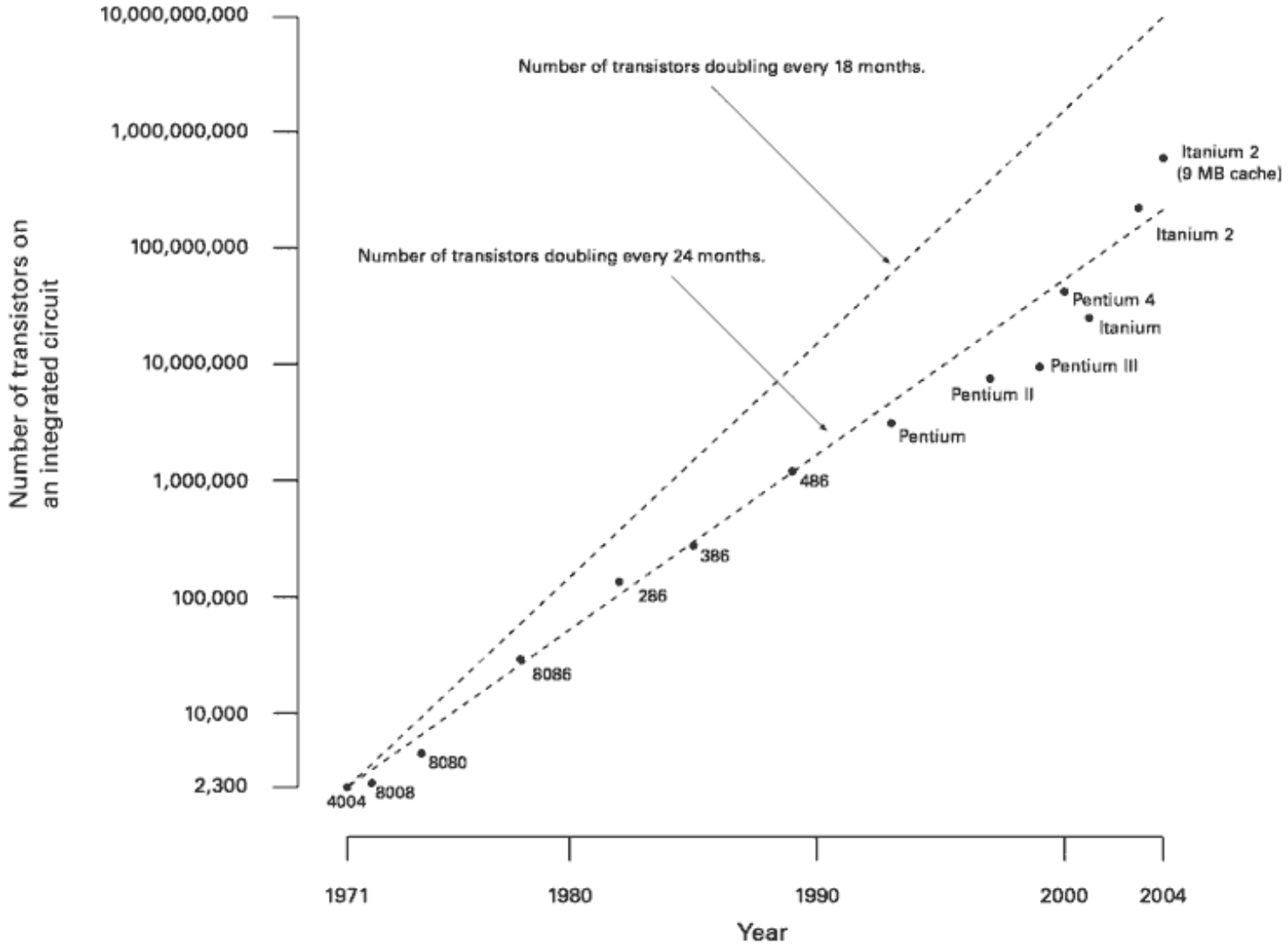


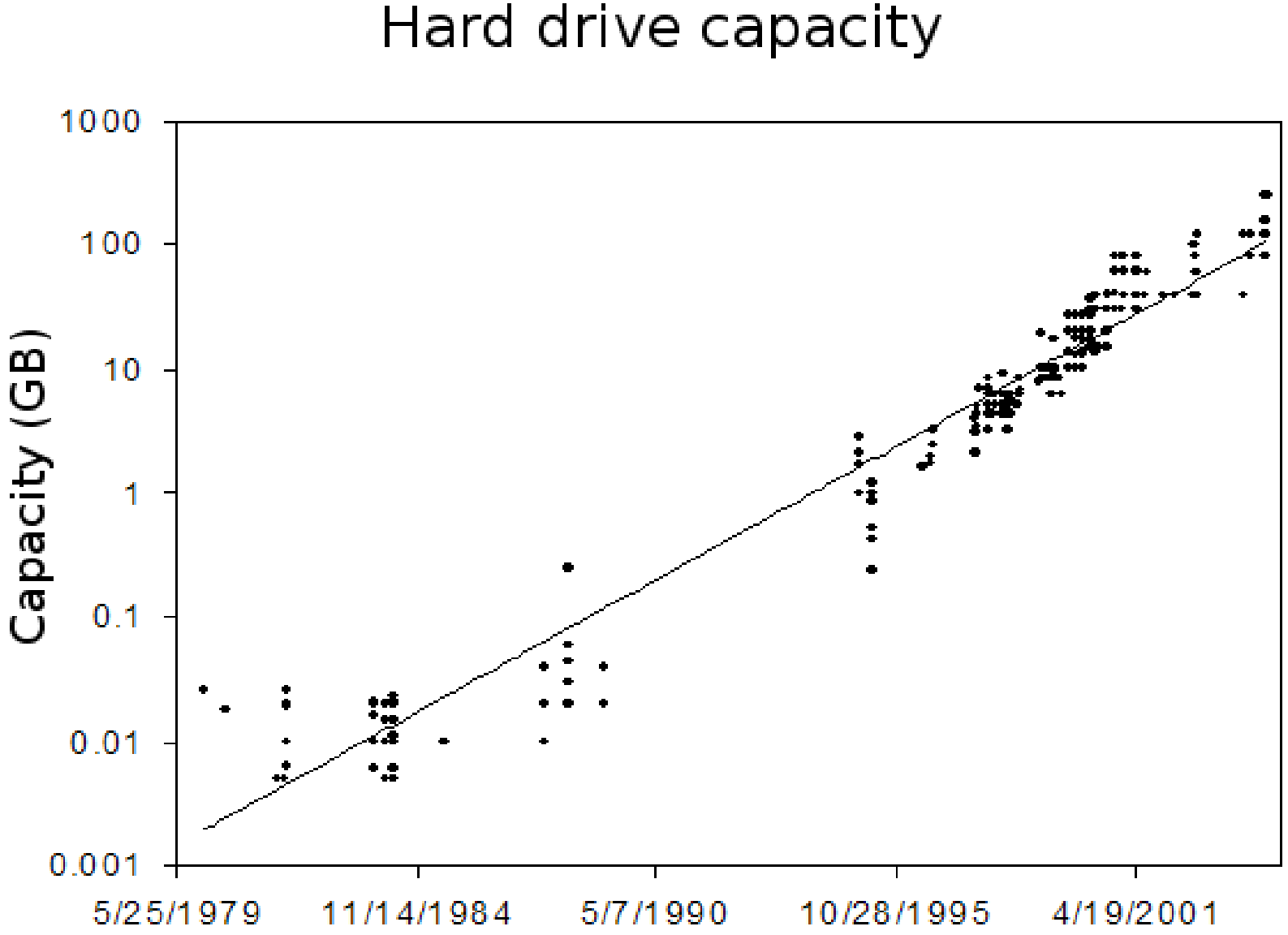
Predicting the future: the last 40 yrs

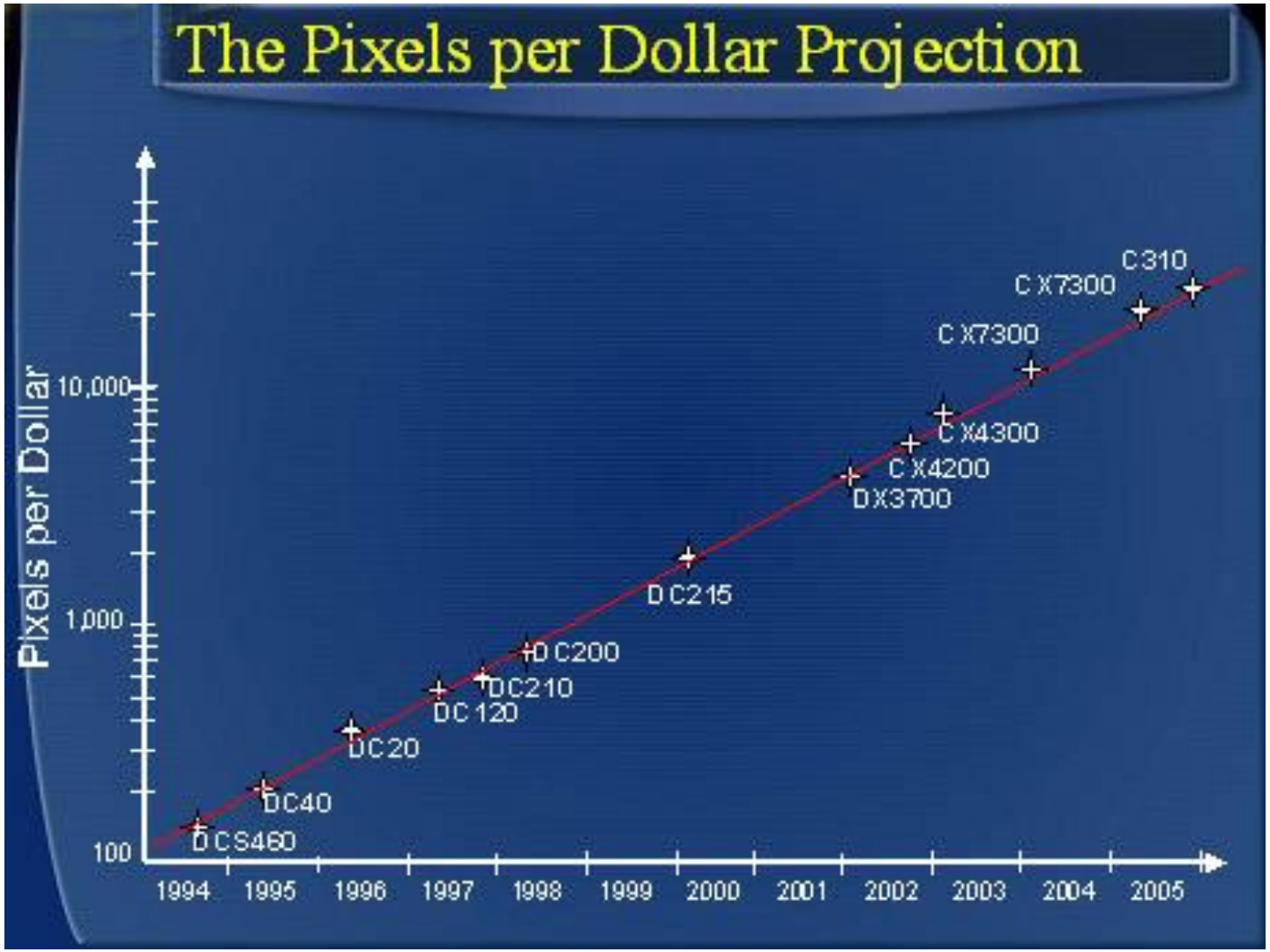


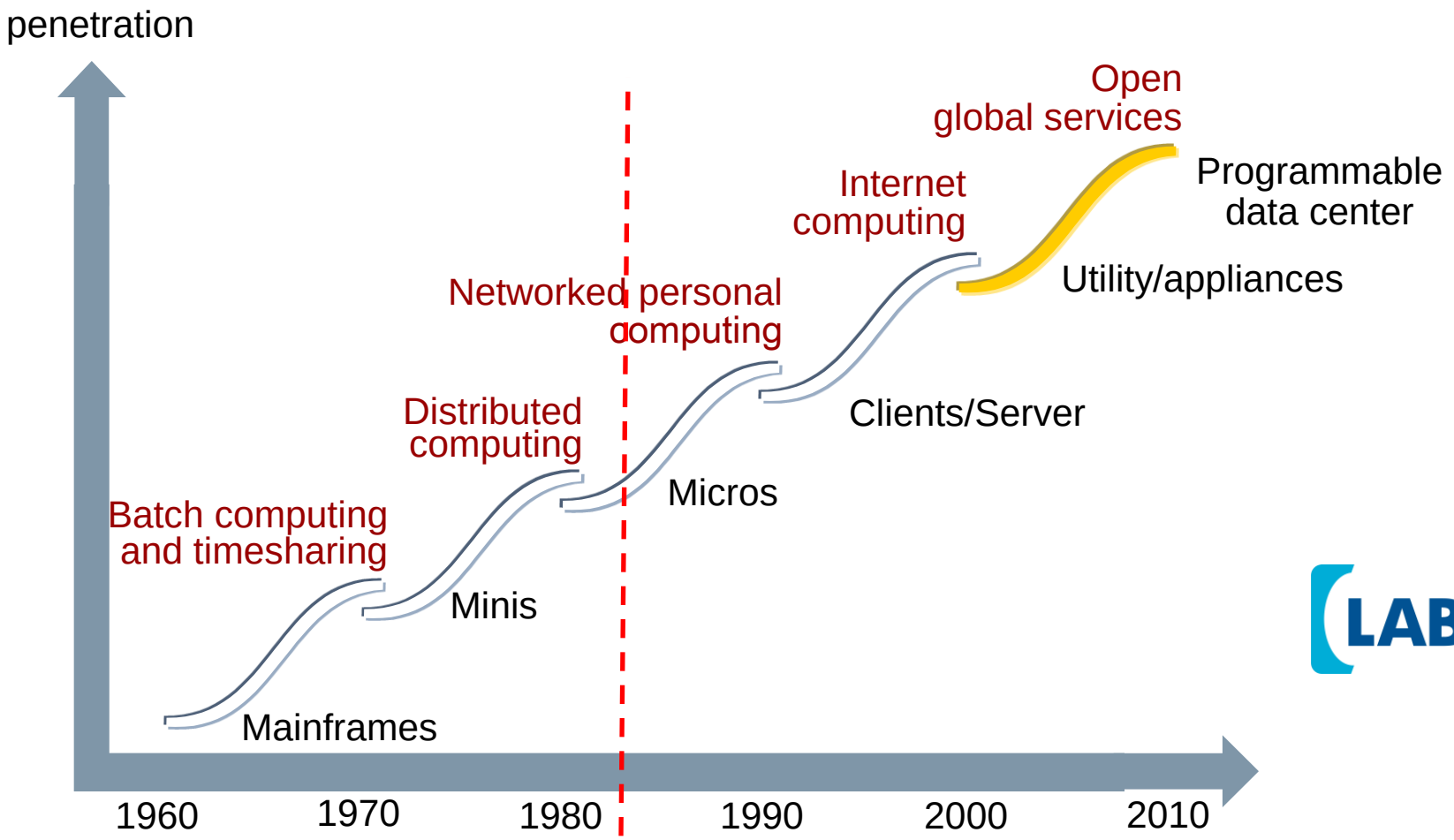

1962
1965
1970
Relative Manufacturing Cost per Component
Number of Components per Integrated Circuit

Moore's Law

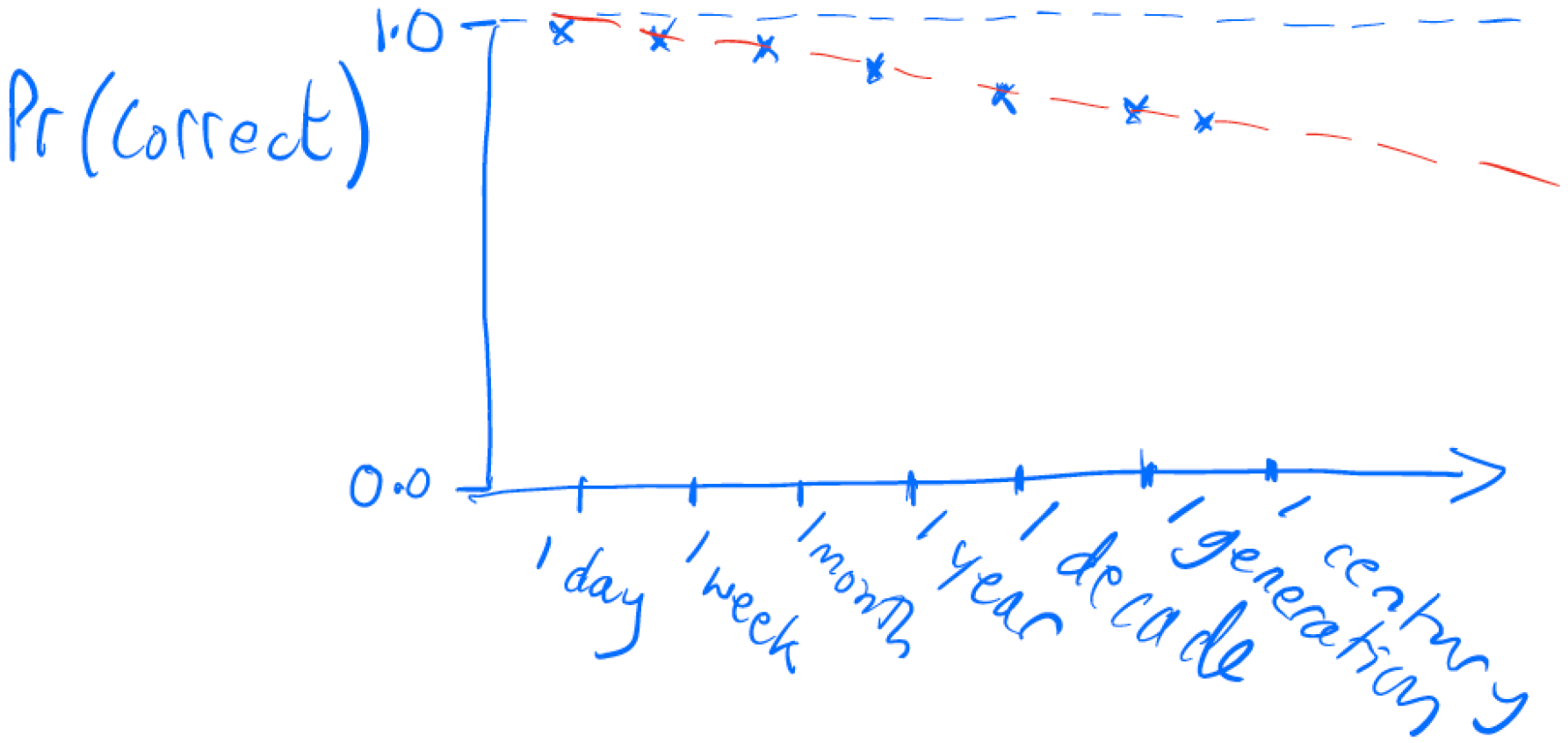

Number of transistors on an integrated circuit
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
Number of transistors doubling every 18 months.
Number of transistors doubling every 24 months.
Itanium 2 (9 MB cache)
Itanium 2
Pentium 4
Itanium
Pentium III
Pentium II
Pentium
486
386
286
8086
8080
8008
4004
1971
1980
1990
2000
2004
Year


Hard drive capacity
Capacity (GB)
1000
100
10
1
0.1
0.01
0.001
5/25/1979
11/14/1984
5/7/1990
10/28/1995
4/19/2001

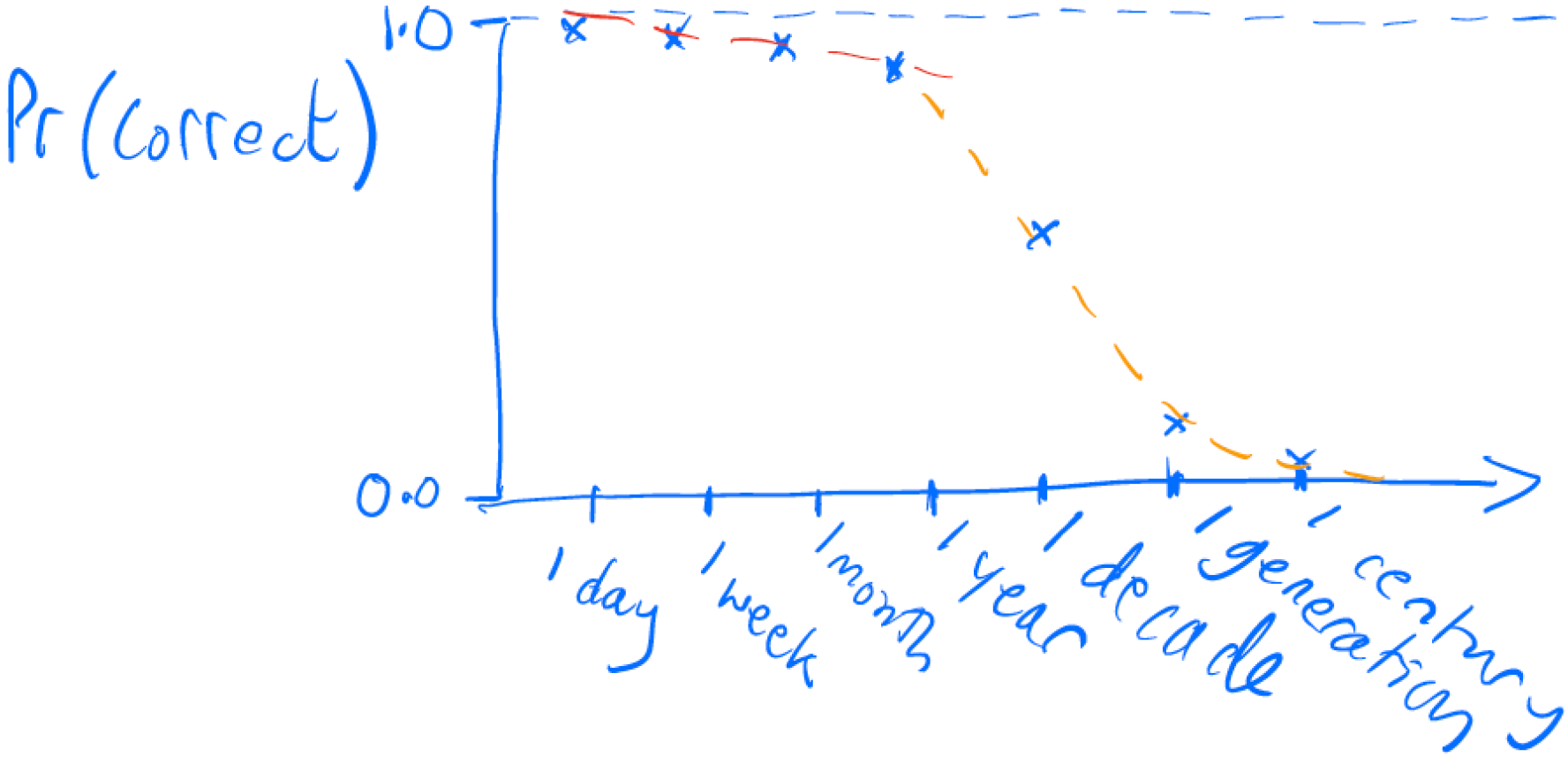

The Pixels per Dollar Projection
Pixels per Dollar
10,000
1,000
100
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
D CS460
DC40
DC20
DC120
DC210
DC200
DC215
DX3700
CX4200
CX4300
CX7300
CX7300
C310

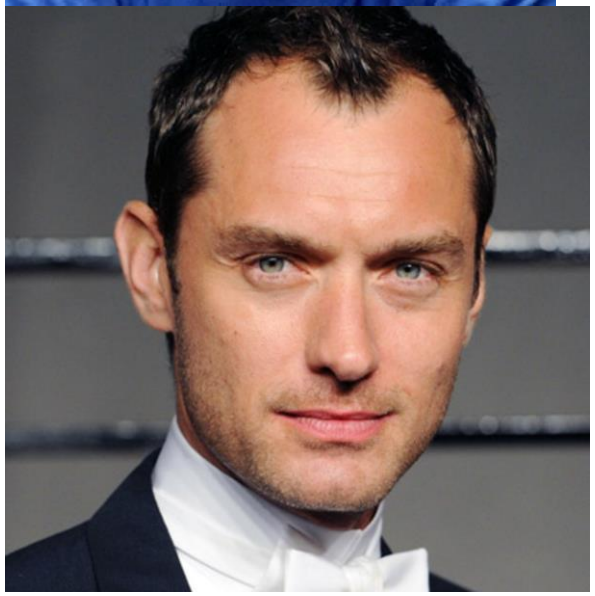

penetration
Open
global services
Internet
computing
Programmable
data center
Networked personal
computing
Utility/appliances
Distributed
computing
Clients/Server
Batch computing
and timesharing
Micros
Minis
Mainframes
1960
1970
1980
1990
2000
2010

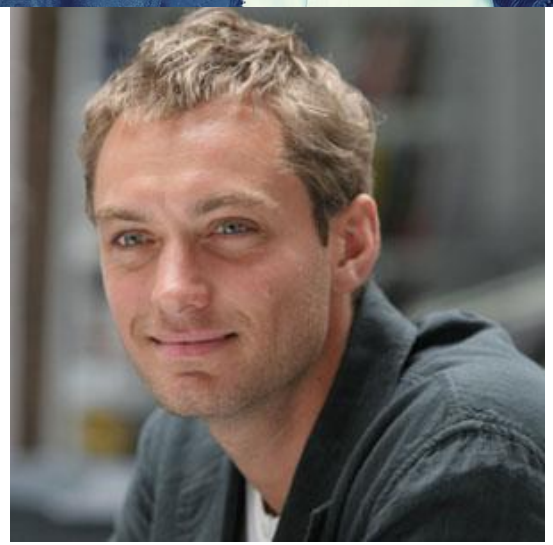
Predicting the future: the last 40 yrs



Predicting the future: the next 40 yrs







ON
OFF

PHYSICS & ASTRONOMY

Campbell's
CONDENSED SOUP SOPA CONDENSADA
TOMATO
TOMATE
NET WT. / PESO NETO 50 OZ.
(3 LB. 2 OZ.) (1.41kg)
SERVING SUGGESTION

ELIZABETH II
D·G·REG·F·D
ONE PENNY
TWO PENCE
FIVE PENCE
TEN PENCE
50
20 PENCE

STOP

Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

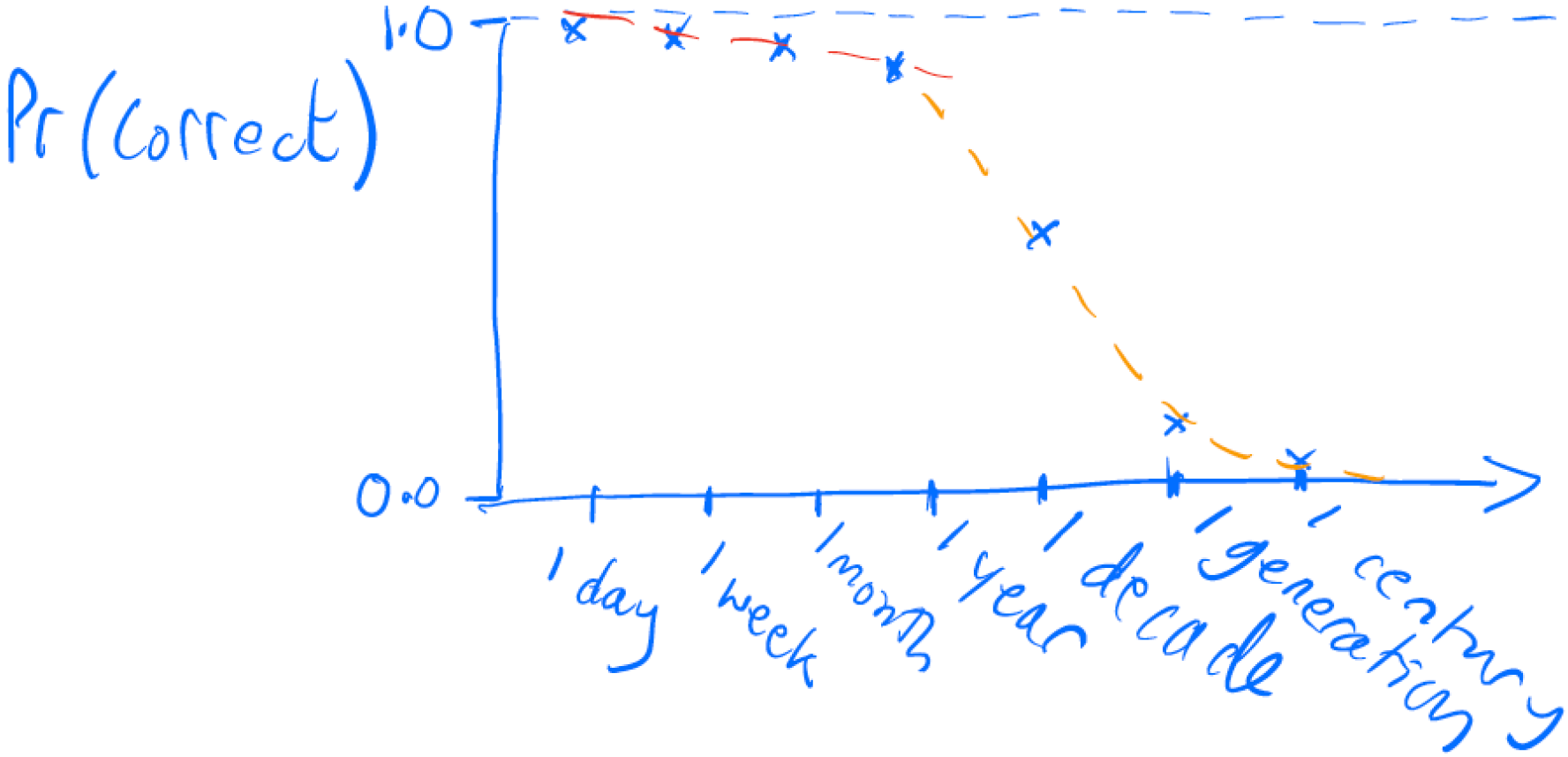
Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

ON
OFF

Campbell's
CONDENSED SOUP
SOPA CONDENSADA
TOMATO
TOMATE

ON
OFF

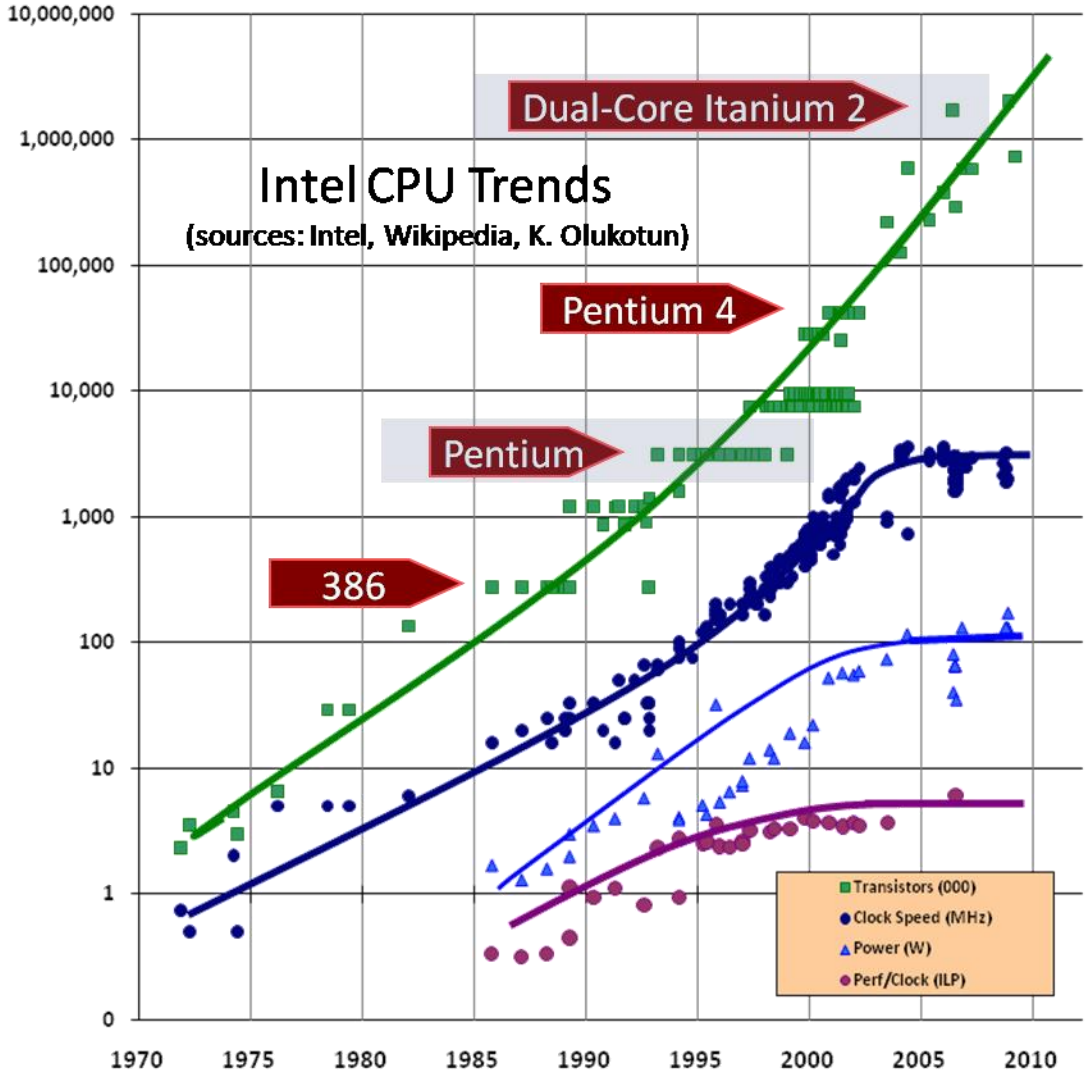
Predicting the future: the next 40 yrs



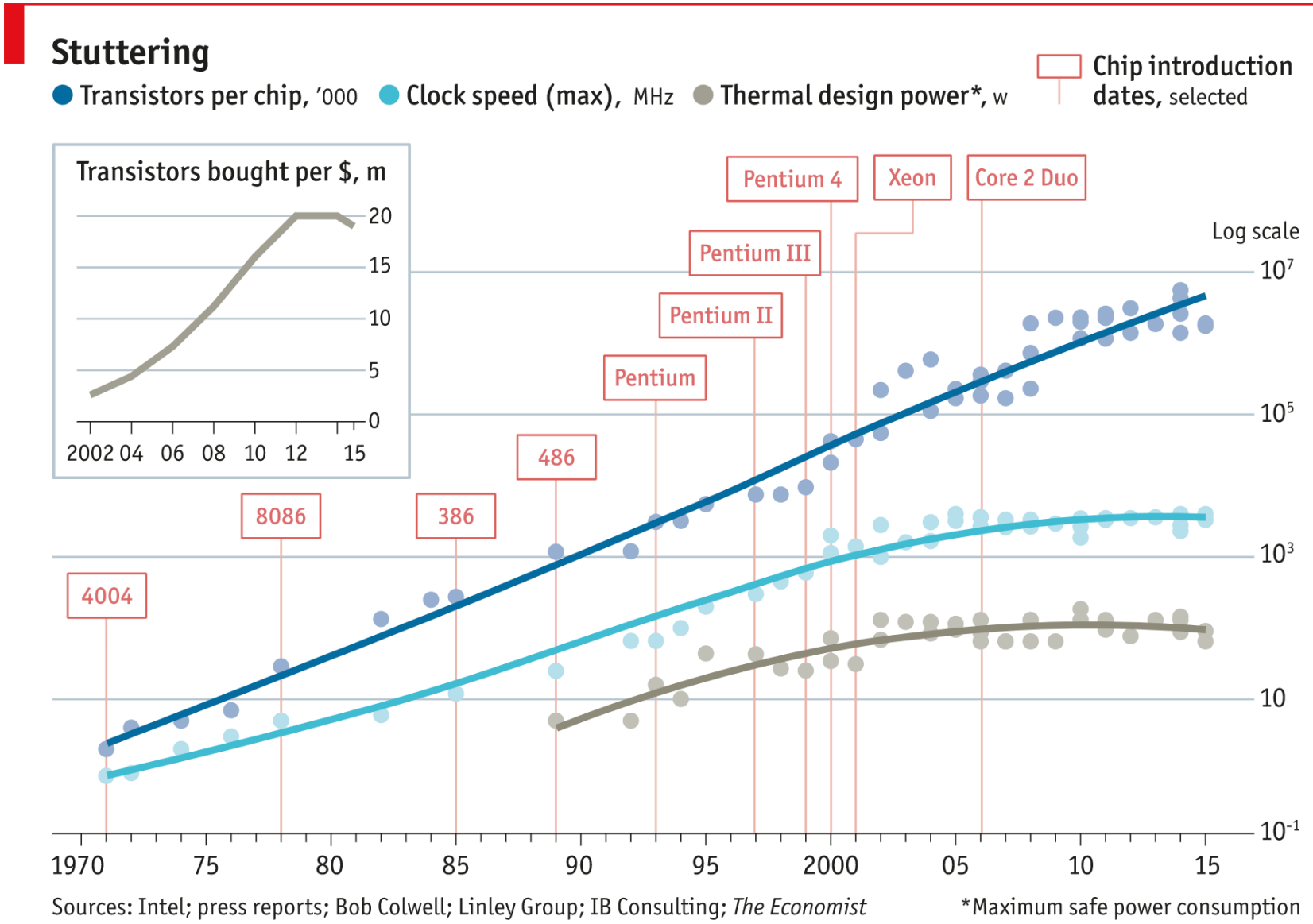
THING 1
THING 2

THING
1

Source: Herb Sutter, “The Free Lunch is Over” *Dr Jobbs’ Journal*, 2005.

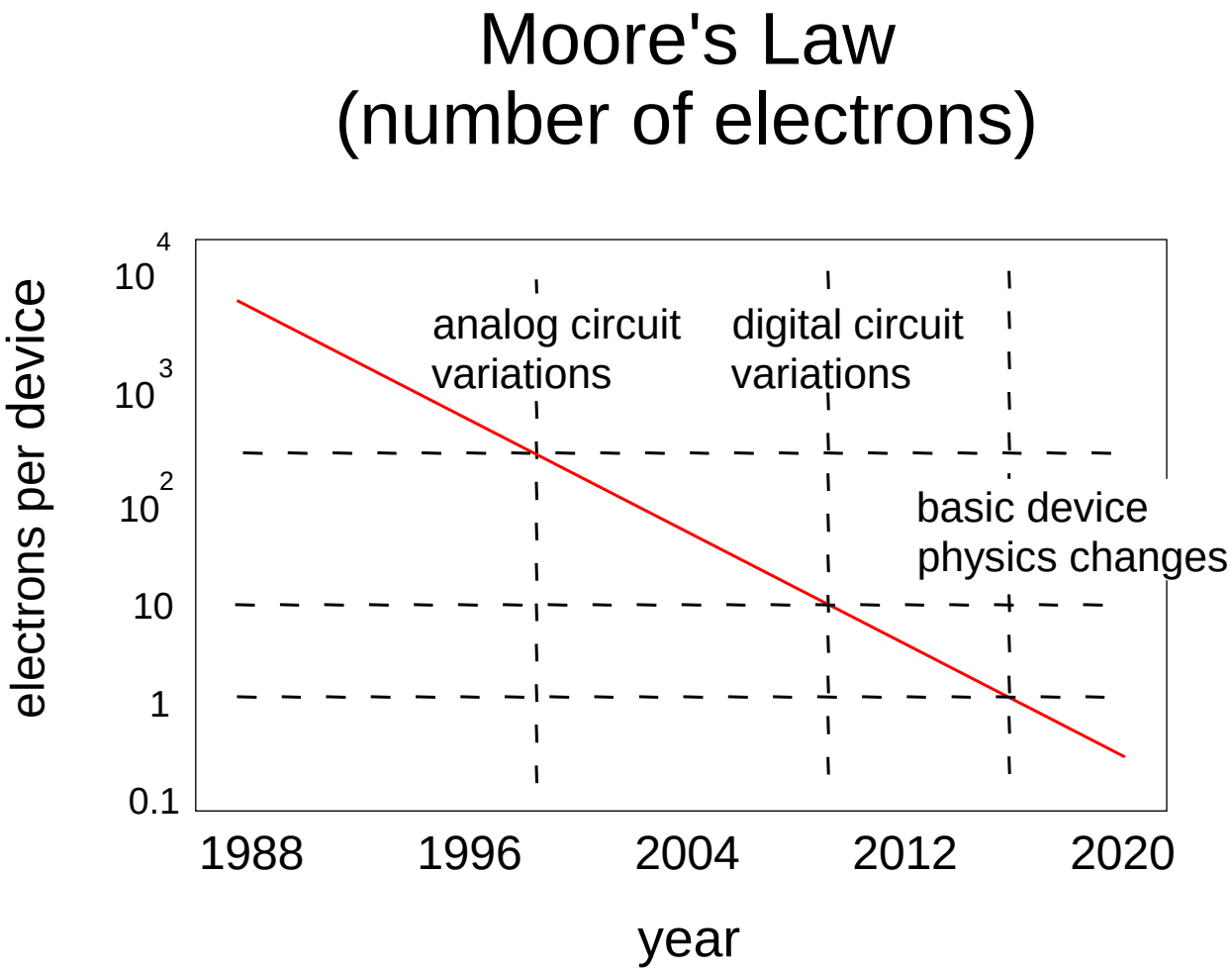


www.economist.com/technology-quarterly/2016-03-12/after-moores-law

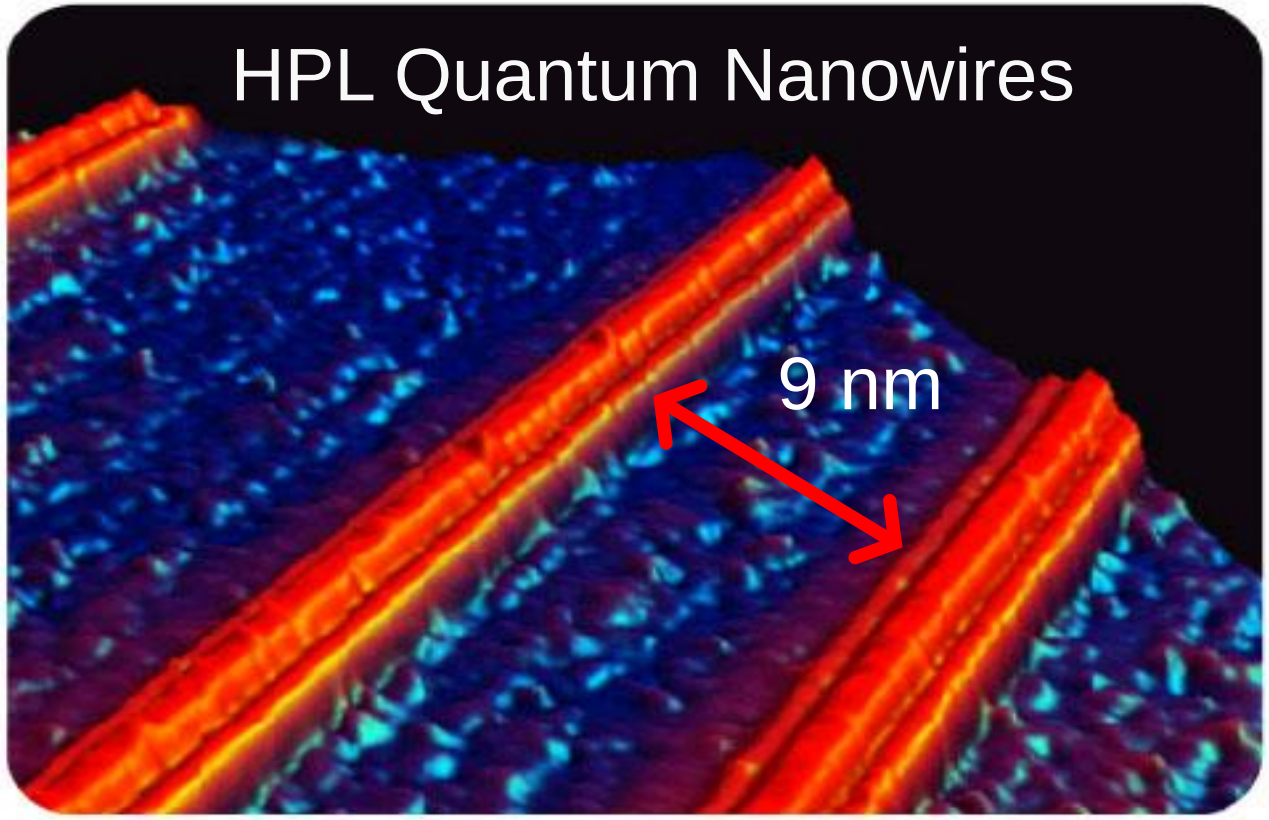


THING
2

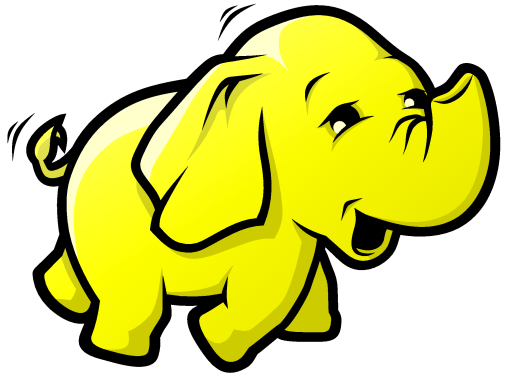
Source: HP Labs c.2004

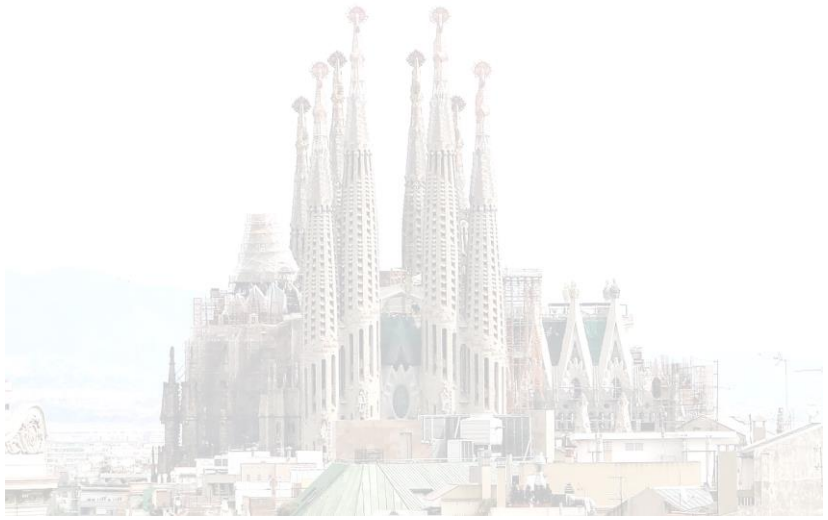


Source: HP Labs c.2004

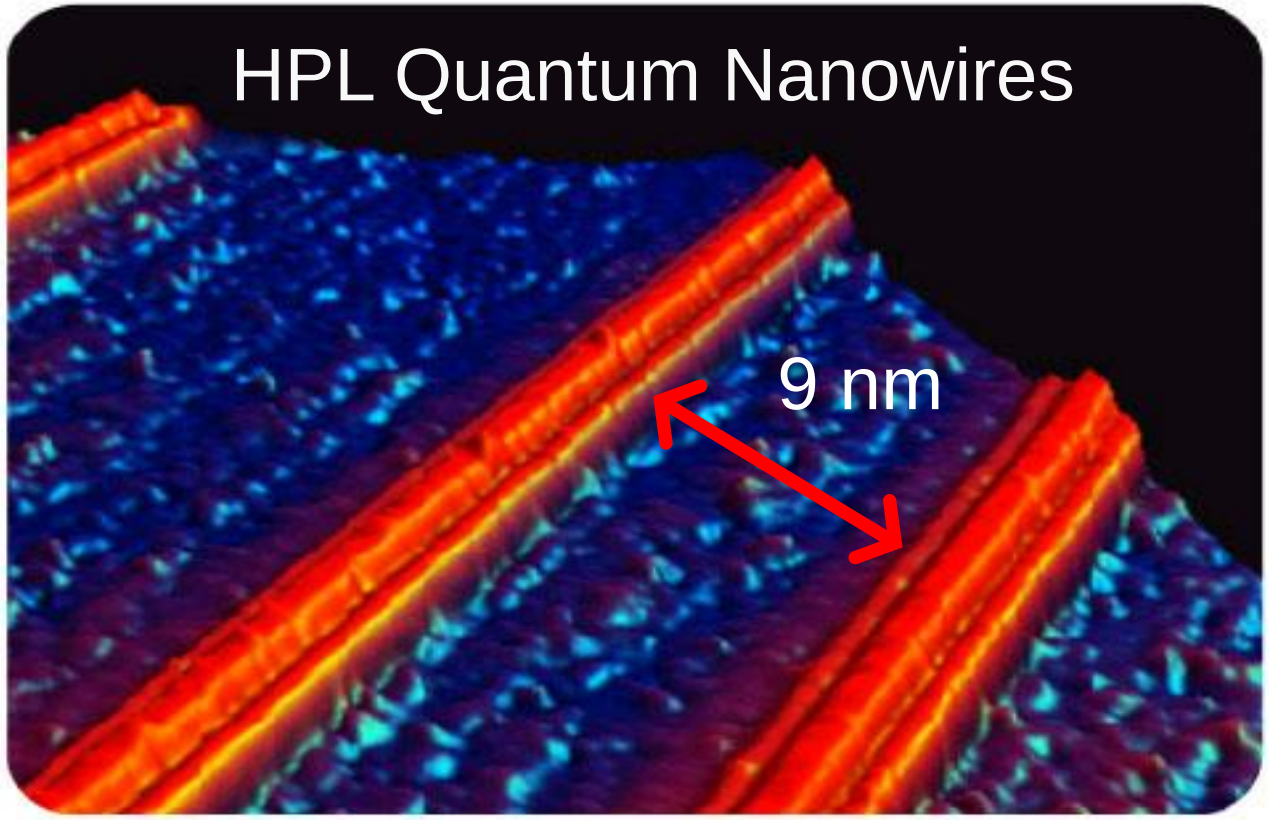


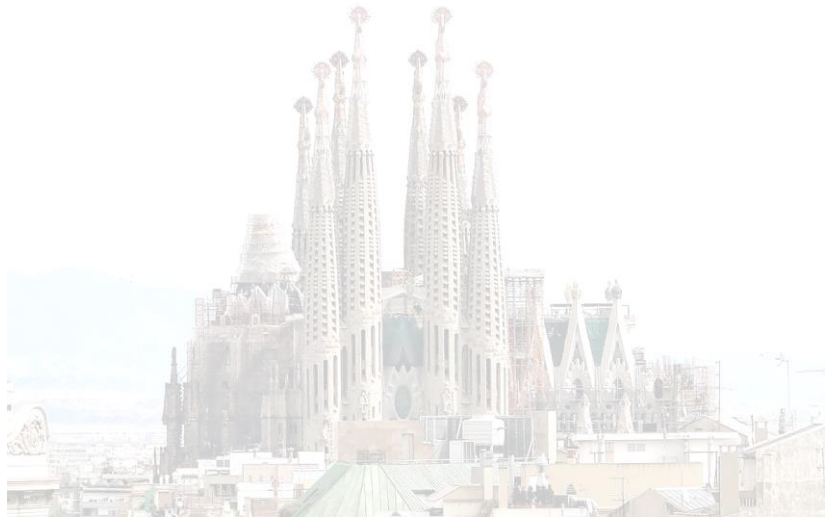
WHERE
NEXT

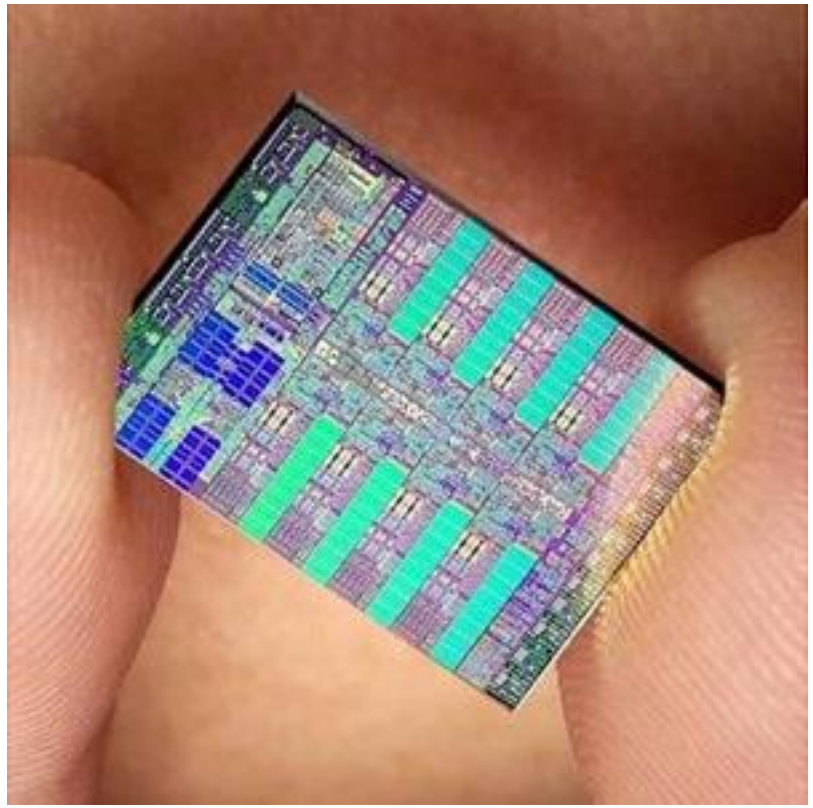


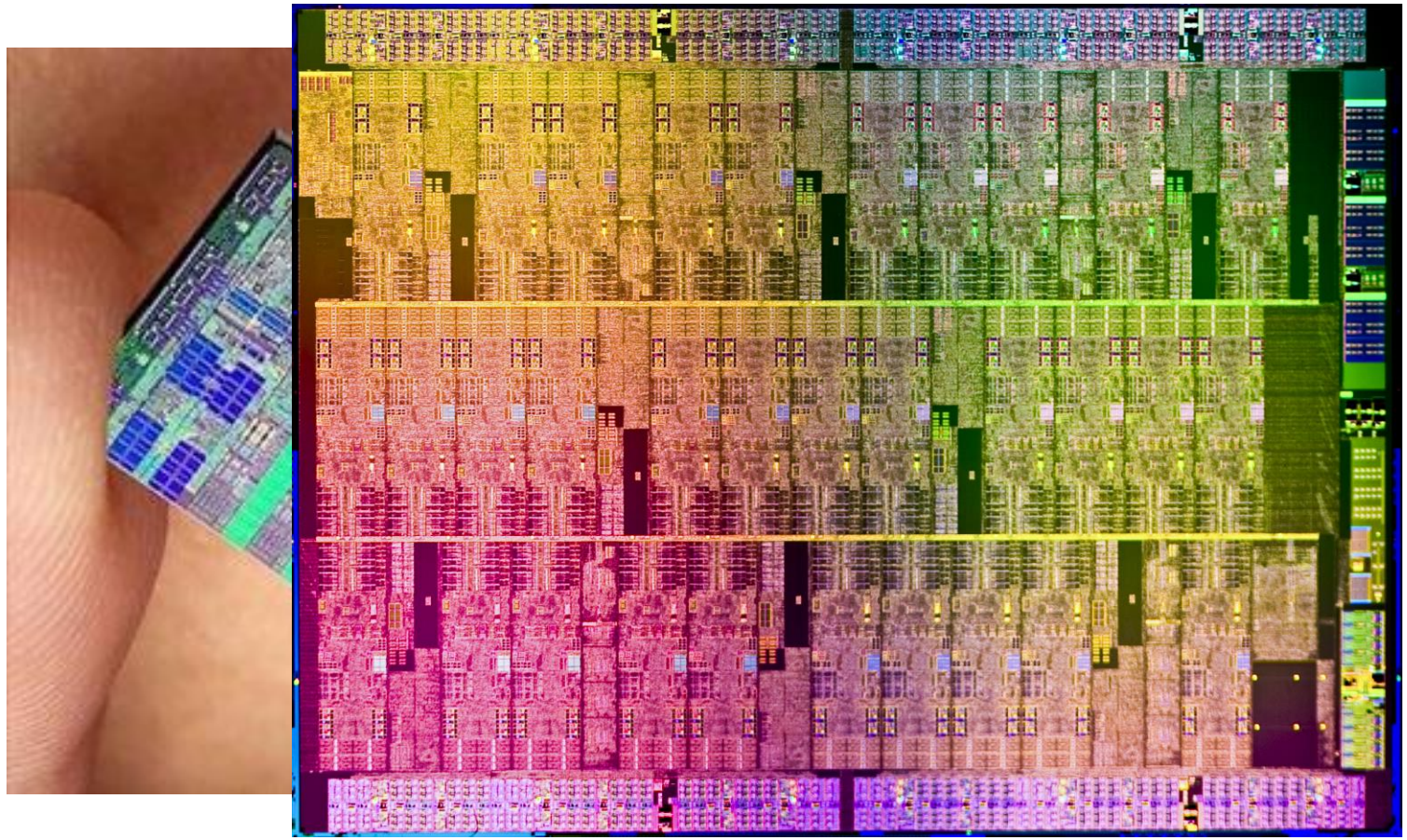


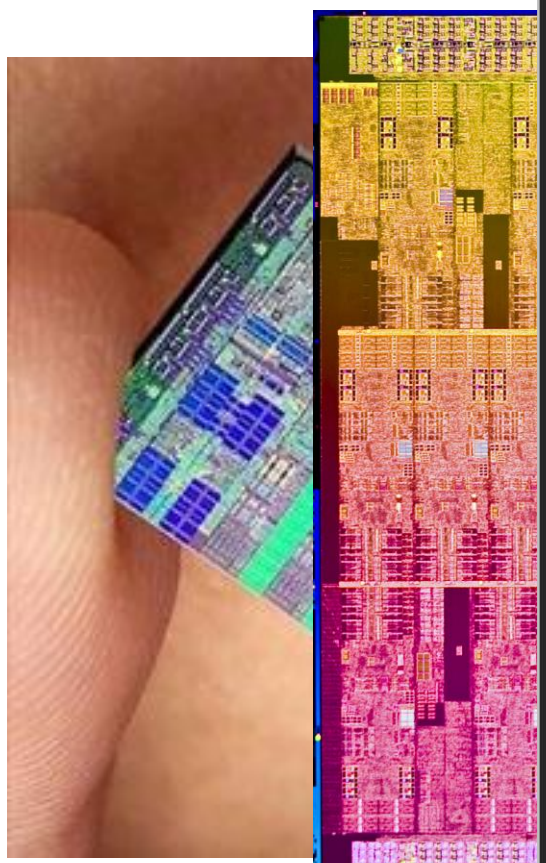
Source: HP Labs c.2004





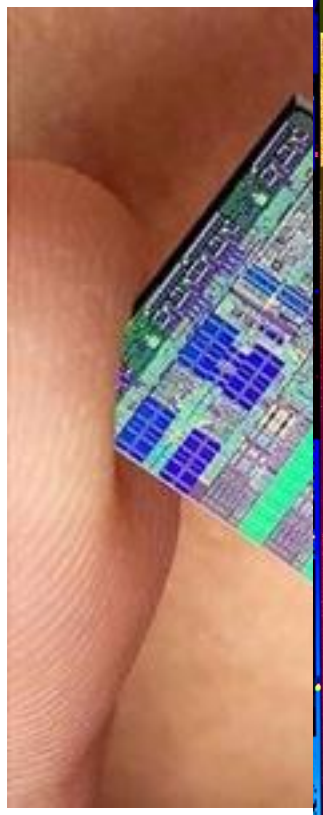


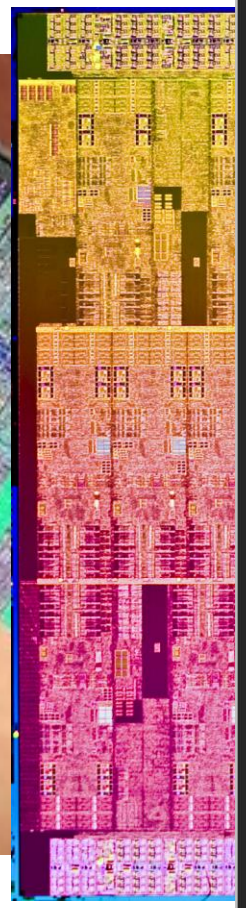


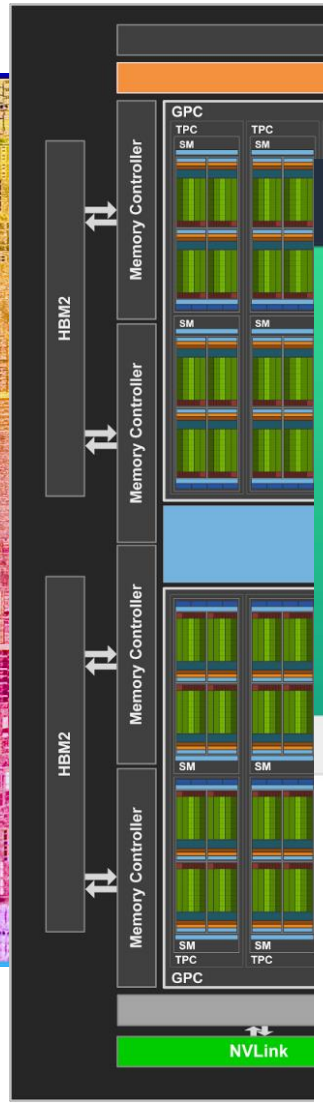


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
HBM2
Memory Controller
NVLink
GTX 1060
NVIDIA







Menu aws Products Solutions Pricing More English My Account Sign In to the Console

Amazon EC2 Elastic GPUs

The most flexible and cost effective way to add graphics acceleration to Amazon EC2 Instances.

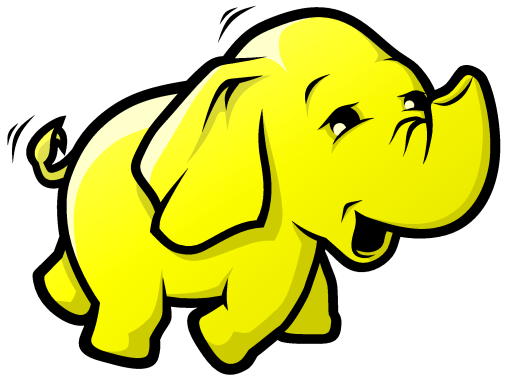
Get started with Amazon EC2 Elastic GPUs

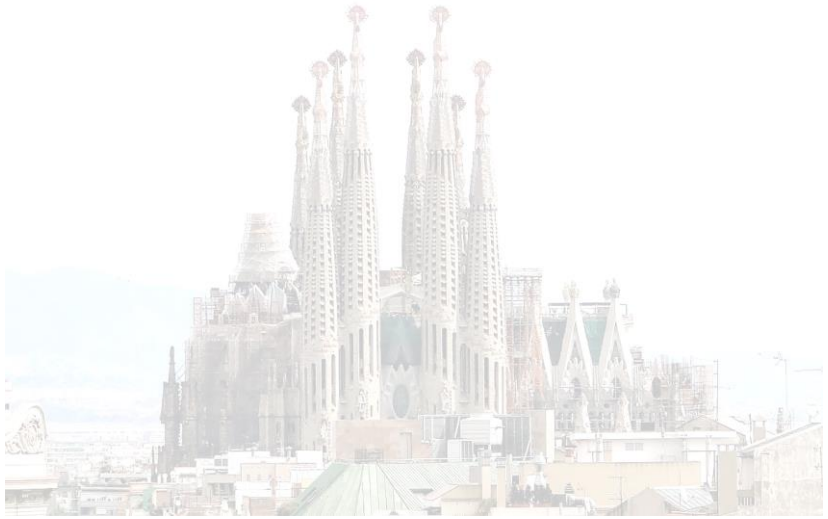
Product Details | Pricing | Getting Started | FAQs | Partners | Documentation

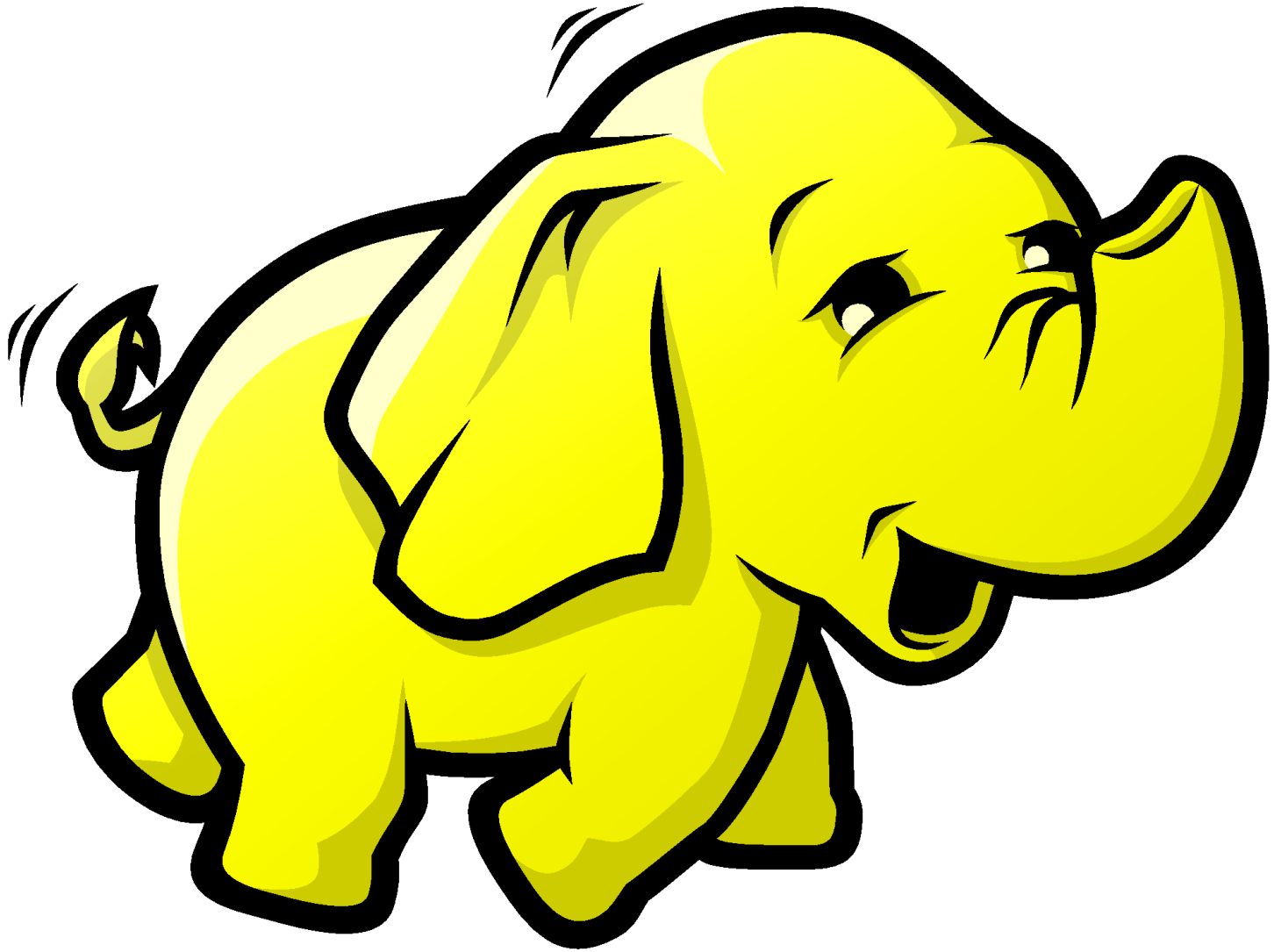
Amazon EC2 Elastic GPUs allow you to easily attach low-cost graphics acceleration to a wide range of EC2 instances over the network. Simply choose an instance with the right amount of compute, memory, and storage for your application, and then use Elastic GPUs to add the GPU resources needed to accelerate the graphics performance of your application for a fraction of the cost of standalone graphics instances.

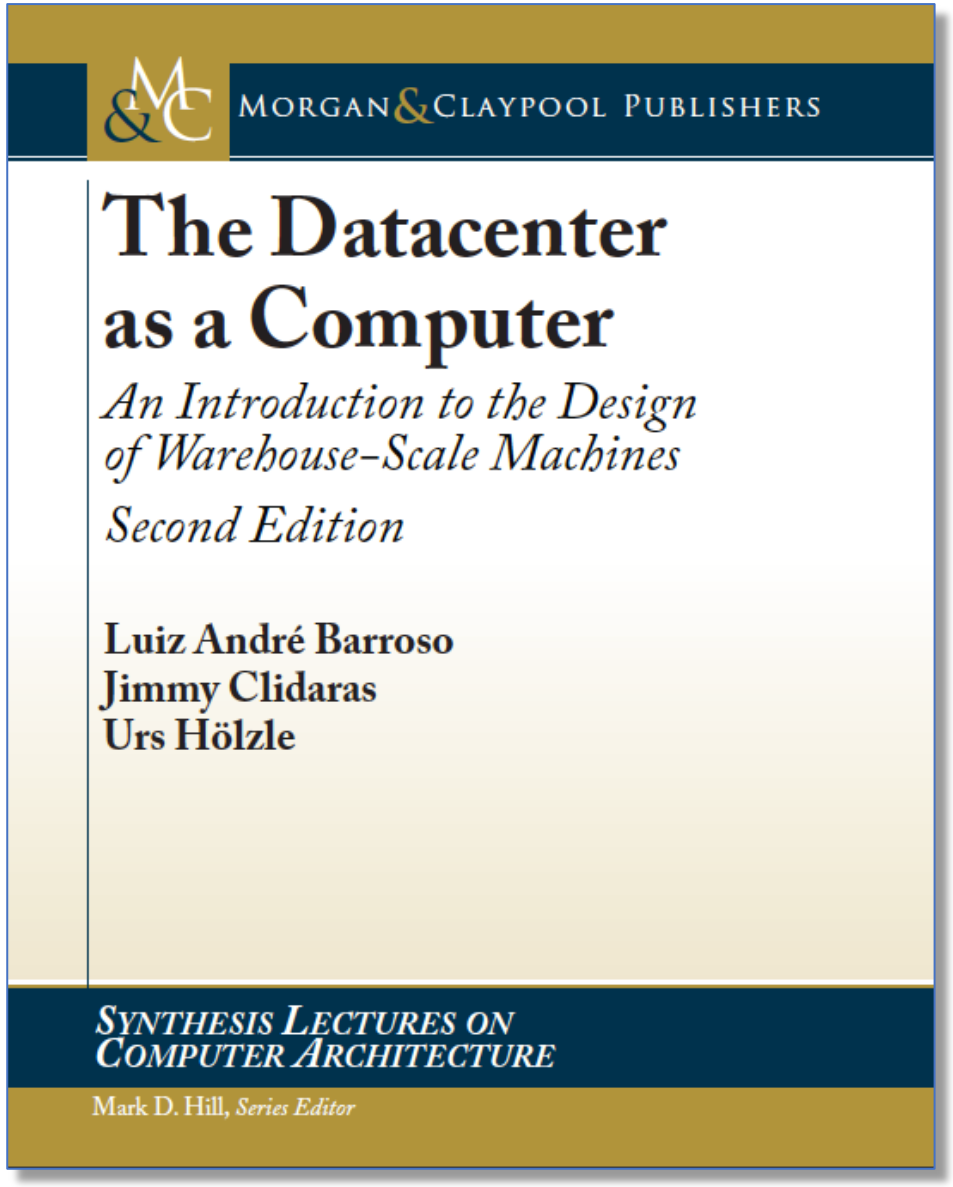
Amazon EC2 Elastic GPUs make it easy to attach graphics acceleration to existing Amazon EC2 instances in much the same way as attaching Amazon EBS volumes. With Elastic GPUs, you can configure the right amount of graphics acceleration to your particular workload without being constrained by fixed hardware configurations and limited GPU selection. Elastic GPUs support OpenGL 4.0 and offer up to 8GiB of GPU memory, making them ideally suited for any workload that needs a small amount of additional GPU such as virtual desktops, gaming, industrial design, or HPC

Introduction to Amazon EC2 Elastic GPUs

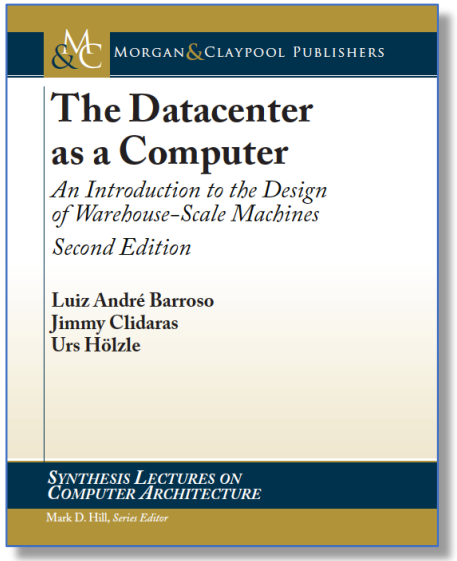


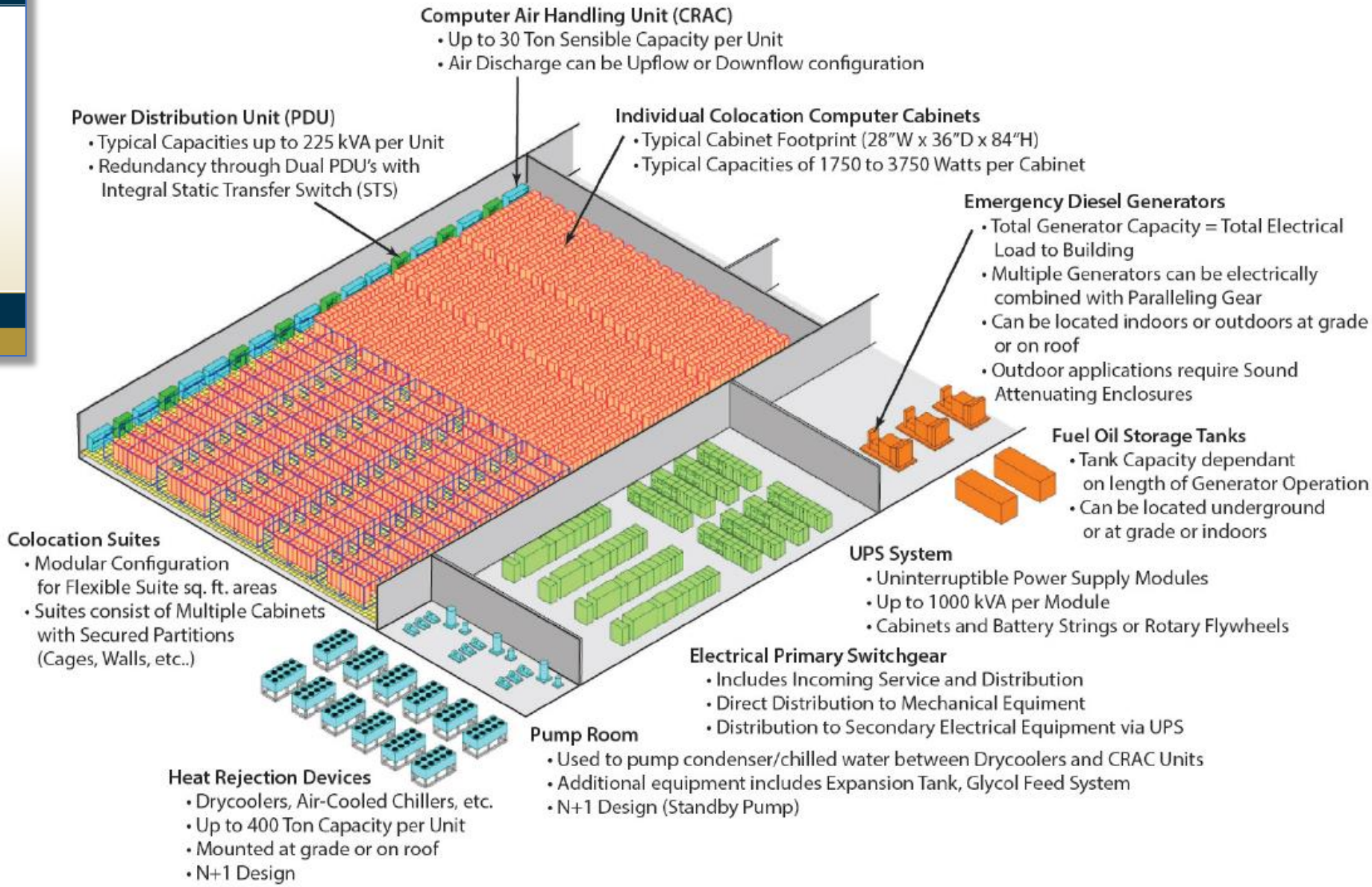


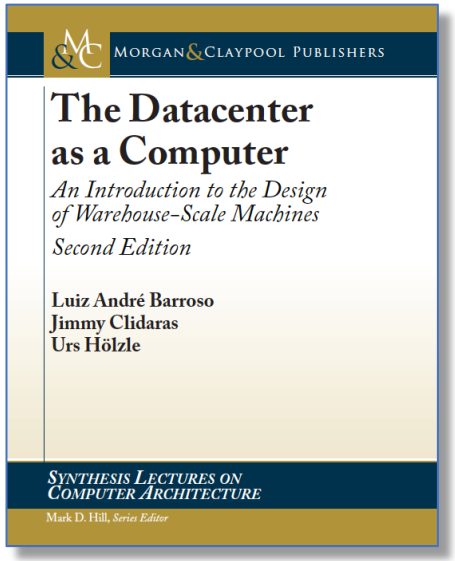




1/e 2009; 2/e 2013


Morgan & Claypool Publishers
The Datacenter as a Computer
An Introduction to the Design of Warehouse-Scale Machines
Second Edition
Luiz André Barroso
Jimmy Clidaras
Urs Hölzle
Synthesis Lectures on Computer Architecture
Mark D. Hill, Series Editor


Computer Air Handling Unit (CRAC)
• Up to 30 Ton Sensible Capacity per Unit
• Air Discharge can be Upflow or Downflow configuration
Power Distribution Unit (PDU)
• Typical Capacities up to 225 kVA per Unit
• Redundancy through Dual PDU's with Integral Static Transfer Switch (STS)
Individual Colocation Computer Cabinets
• Typical Cabinet Footprint (28"W x 36"D x 84"H)
• Typical Capacities of 1750 to 3750 Watts per Cabinet
Emergency Diesel Generators
• Total Generator Capacity = Total Electrical Load to Building
• Multiple Generators can be electrically combined with Paralleling Gear
• Can be located indoors or outdoors at grade or on roof
• Outdoor applications require Sound Attenuating Enclosures
Fuel Oil Storage Tanks
• Tank Capacity dependant on length of Generator Operation
• Can be located underground or at grade or indoors
Colocation Suites
• Modular Configuration for Flexible Suite sq. ft. areas
• Suites consist of Multiple Cabinets with Secured Partitions (Cages, Walls, etc..)
UPS System
• Uninterruptible Power Supply Modules
• Up to 1000 kVA per Module
• Cabinets and Battery Strings or Rotary Flywheels
Electrical Primary Switchgear
• Includes Incoming Service and Distribution
• Direct Distribution to Mechanical Equiment
• Distribution to Secondary Electrical Equipment via UPS
Pump Room
• Used to pump condenser/chilled water between Drycoolers and CRAC Units
• Additional equipment includes Expansion Tank, Glycol Feed System
• N+1 Design (Standby Pump)
Heat Rejection Devices
• Drycoolers, Air-Cooled Chillers, etc.
• Up to 400 Ton Capacity per Unit
• Mounted at grade or on roof
• N+1 Design



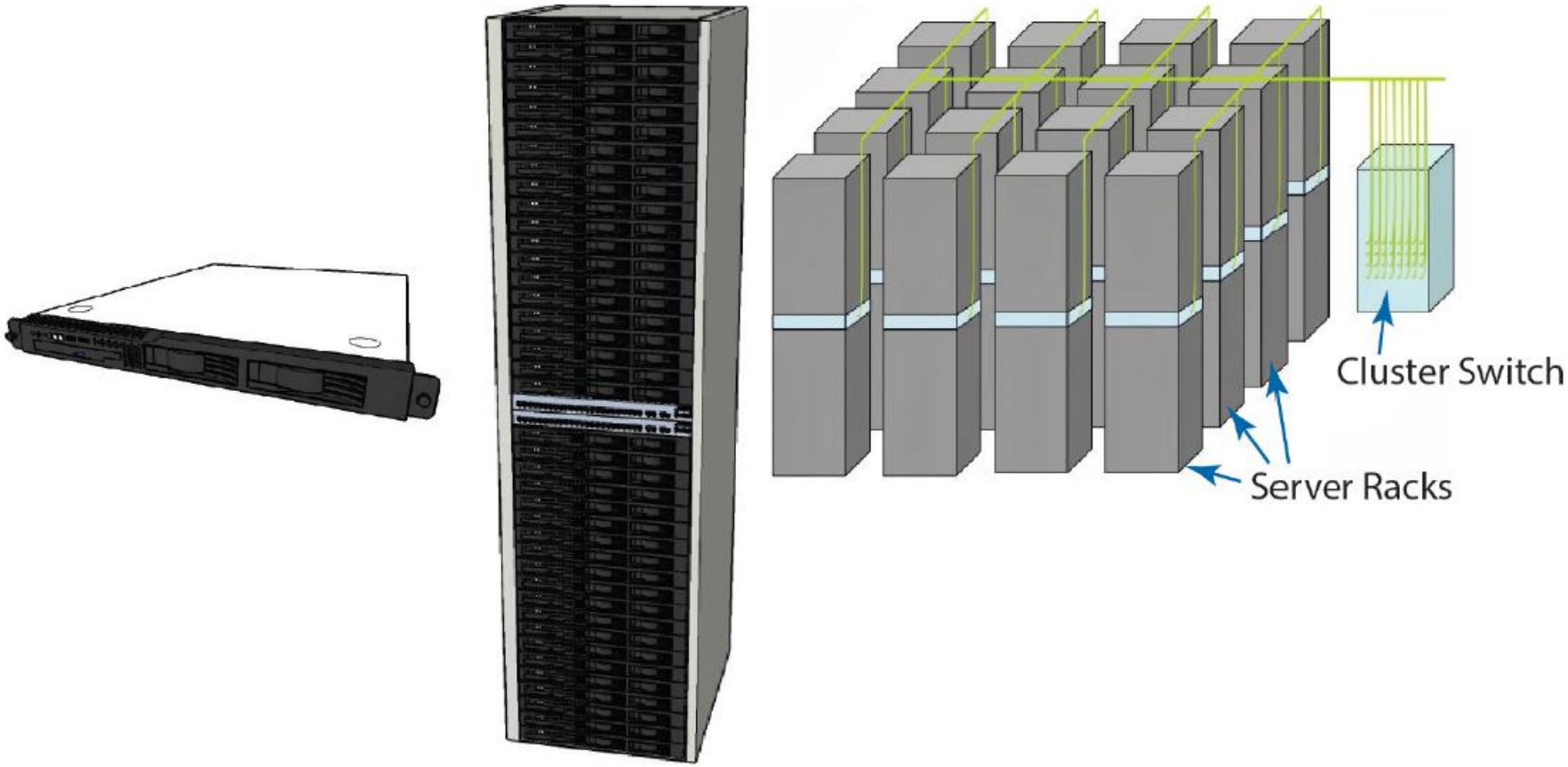
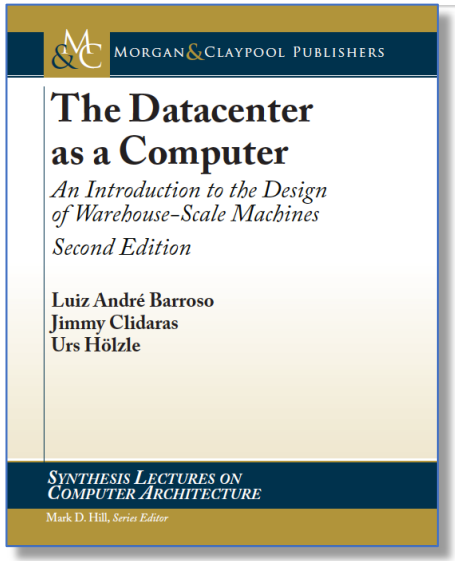


Figure 1.1: Sketch of the typical elements in warehouse-scale systems: 1U server (left), 7' rack with Ethernet switch (middle), and diagram of a small cluster with a cluster-level Ethernet switch/ router (right).



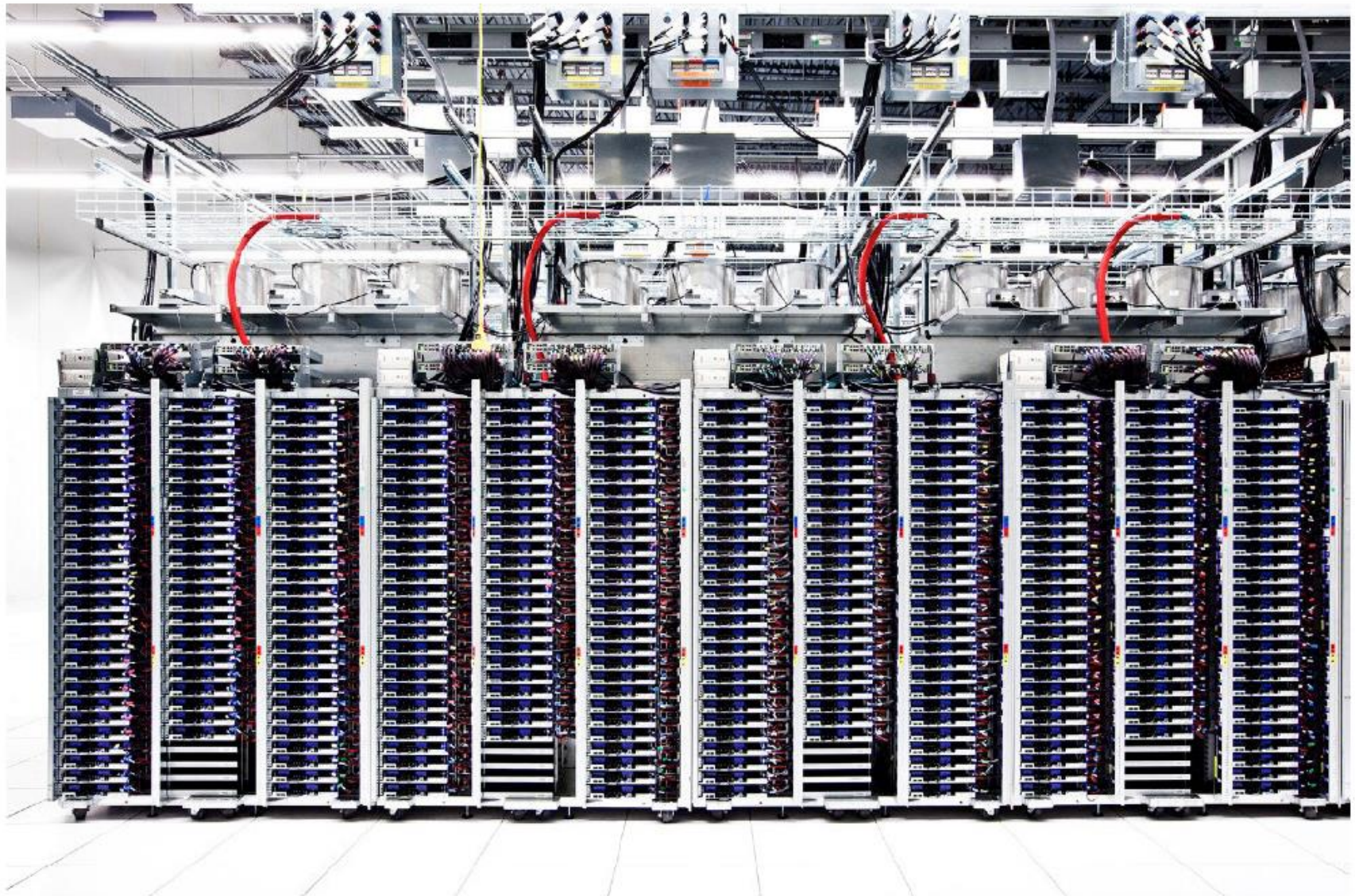
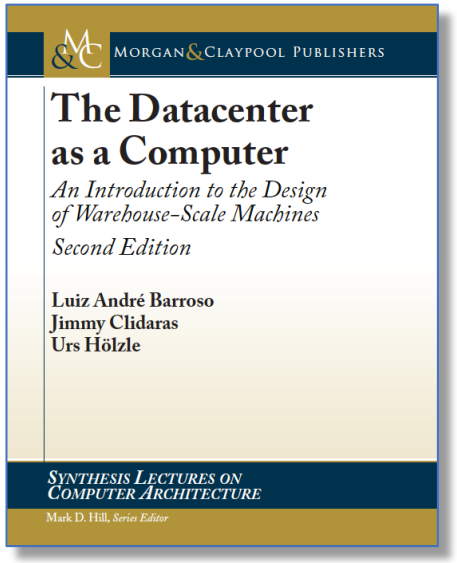


Figure 1.2: Picture of a row of servers in a Google WSC, 2012.



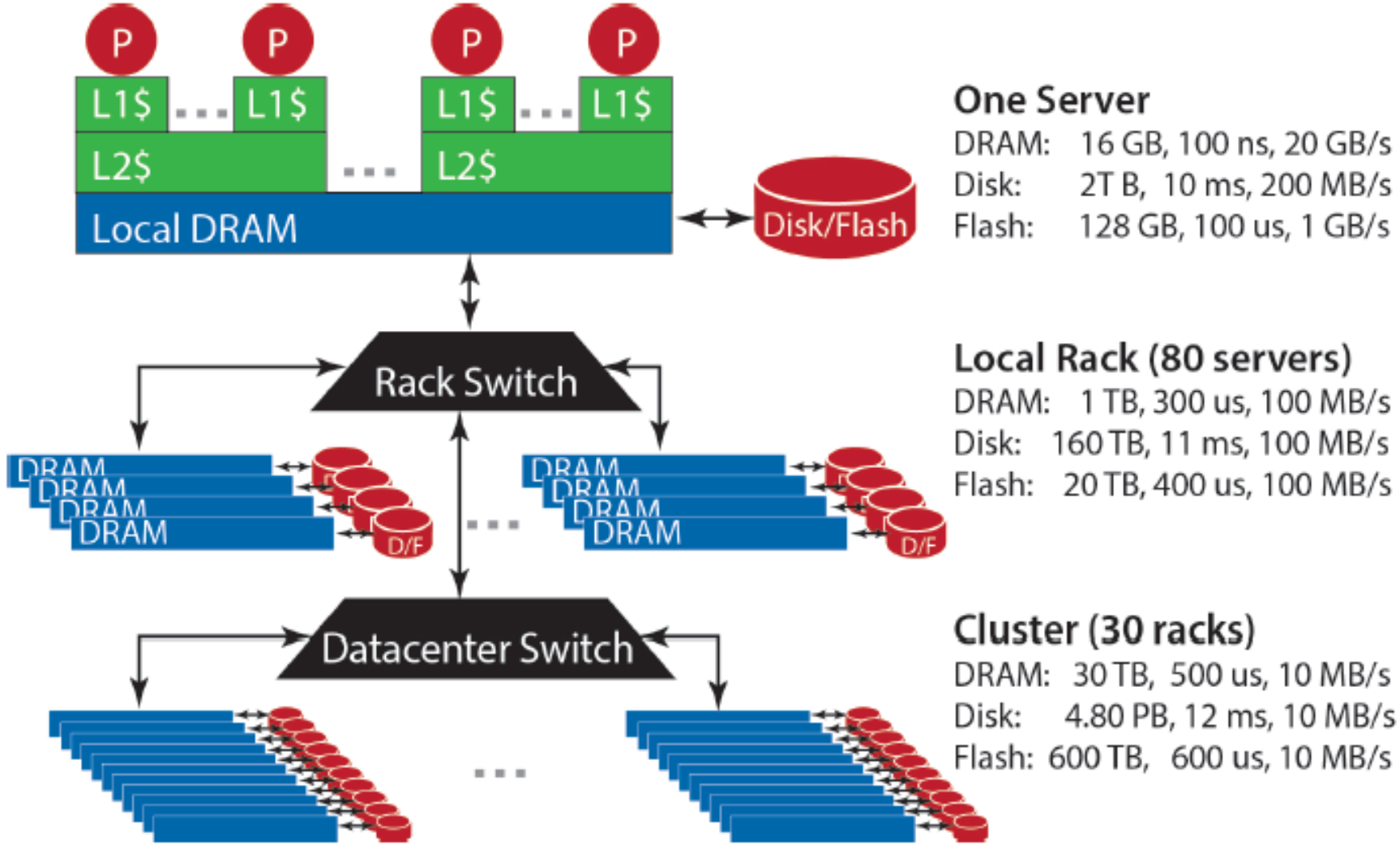
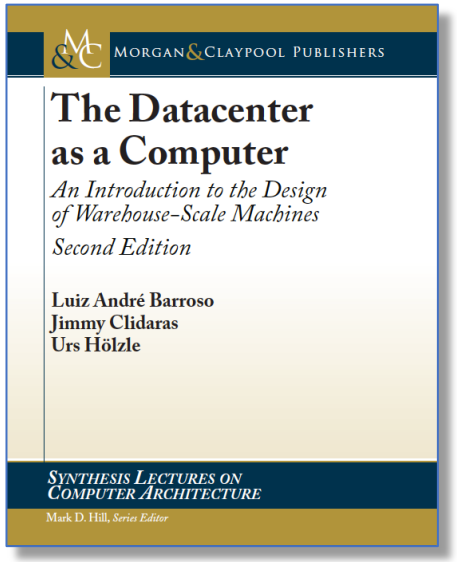


Figure 1.3: Storage hierarchy of a WSC.



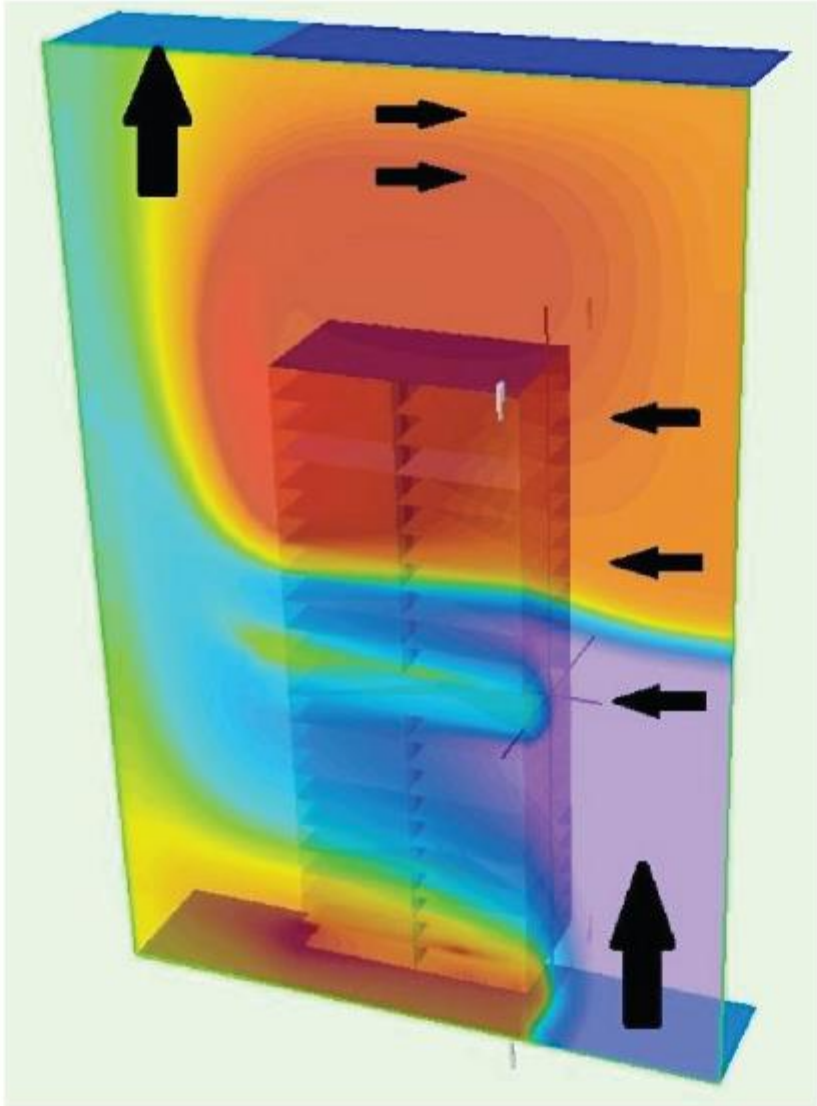
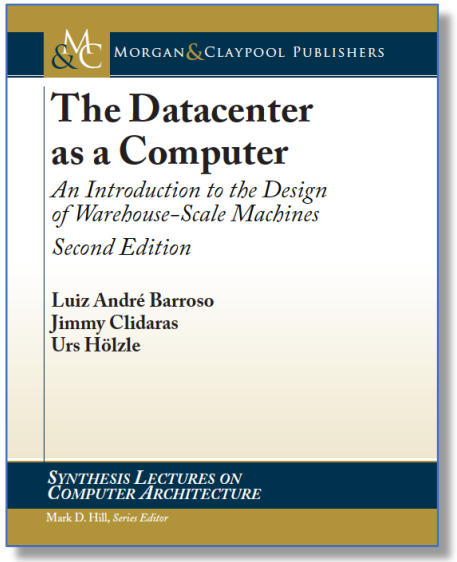


Figure 4.10: CFD model showing recirculation paths and temperature stratification for a rack with under-provisioned airflow.



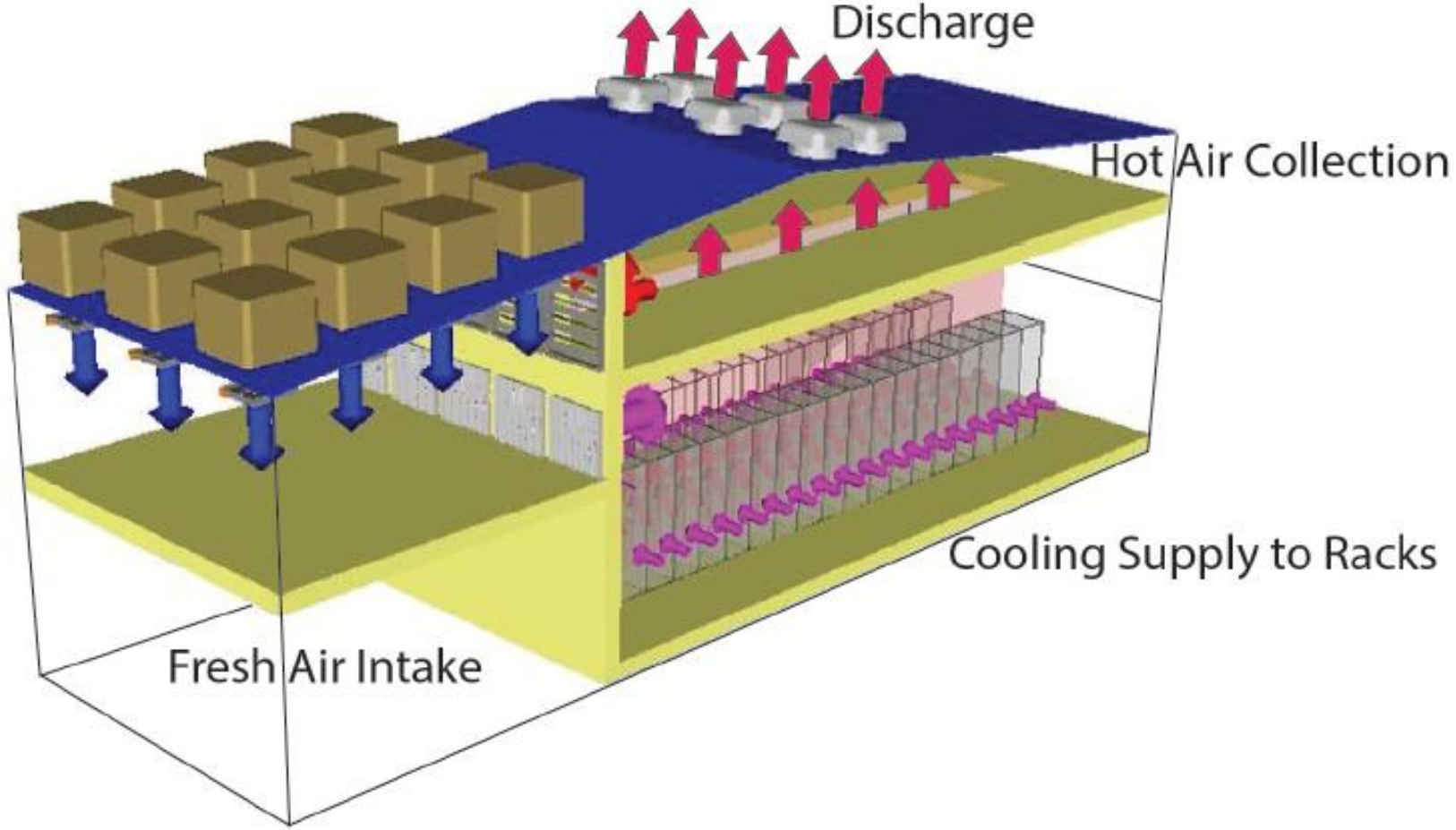
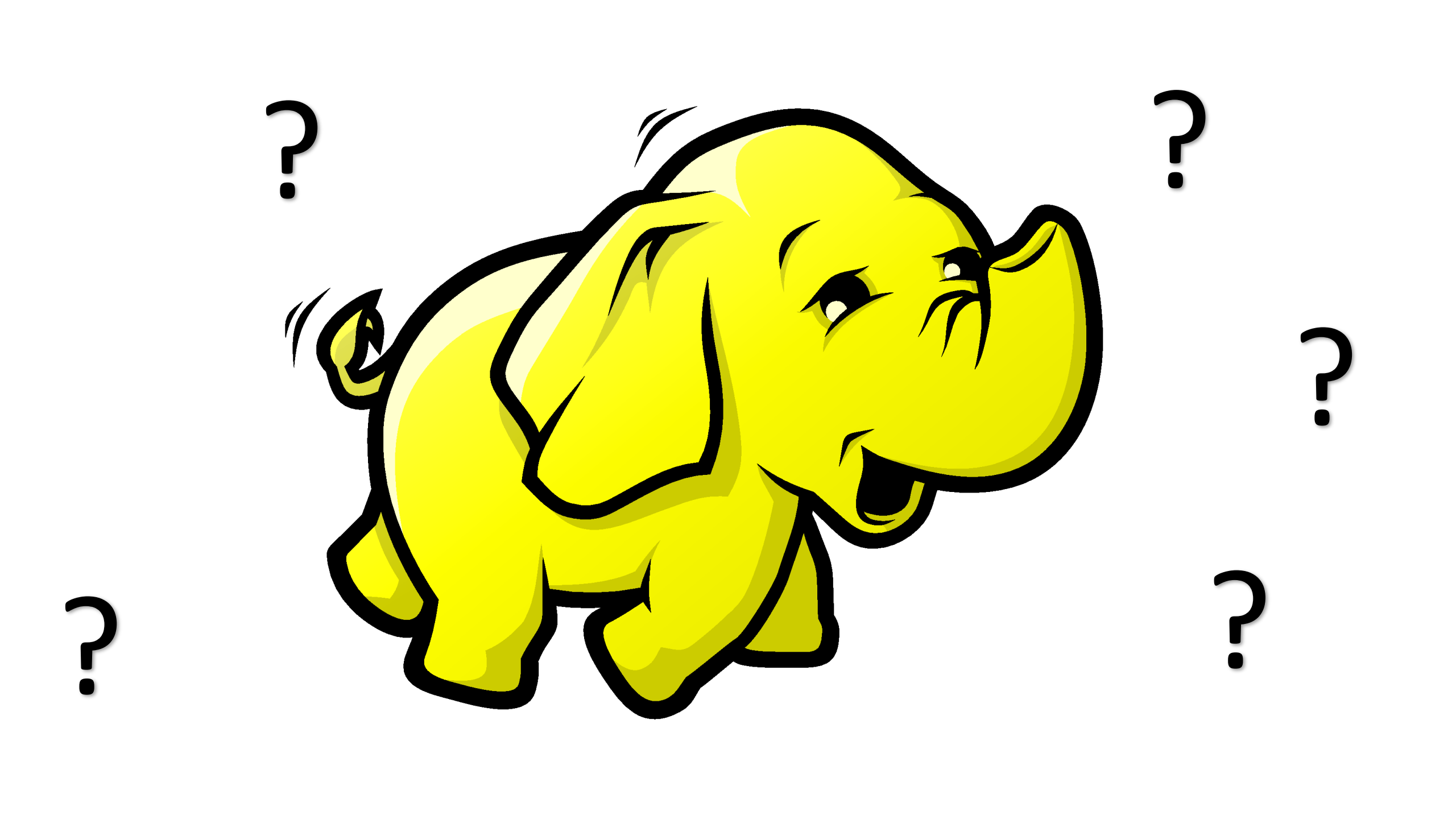


Figure 4.5: Airflow schematic of an air-economized datacenter (source: Perspectives, James Hamilton's blog).



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application

ACM SIGOPS Operating Systems Review. 37(5) 2003

MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The programming model can also be used to parallelize computations across multiple cores of the same machine.

Section 2 describes the basic programming model and gives several examples. In Section 3, we describe an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. In Section 6, we explore the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges these values together to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudocode.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most of the company's products. He is based at Google's Mountain View, CA, facility.

ak Leung

ABSTRACT

We have design
tem, a scalable
data-intensive ap
running on inexp
high aggregate p
While sharing
tributed file syst
vations of our ap
ronment, both cu
departure from
has led us to ree
ically different d
The file syste
It is widely depl
for the generatio
vice as well as research and development efforts that require
large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on
over a thousand machines, and it is concurrently accessed

ION

d implemented the Google File Sys-
rapidly growing demands of Google's
GFS shares many of the same goals
l file systems such as performance,
nd availability. However, its design
observations of our application work-
environment, both current and an-
marked departure from some earlier
nptions. We have reexamined tradi-
ored radically different points in the

lures are the norm rather than the
ystem consists of hundreds or even
achines built from inexpensive com-
ccessed by a comparable number of
quantity and quality of the compo-
nents virtually guarantee that some are not functional at
any given time and some will not recover from their cur-
rent failures. We have seen problems caused by application

Comms. ACM, 51(1): 107-113, January 2008.

Bigtable: A Distributed Storage System for Structured Data

FAY CHANG, JEFFREY DEAN, SANJAY GHEMAWAT, WILSON C. HSIEH, DEBORAH A. WALLACH, MIKE BURROWS, TUSHAR CHANDRA, ANDREW FIKES, and ROBERT E. GRUBER
Google, Inc.

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this article, we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed databases*

General Terms: Design

Additional Key Words and Phrases: Large-Scale Distributed Storage

ACM Reference Format:
Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2008. Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI = 10.1145/1365815.1365816. http://doi.acm.org/10.1145/1365815.1365816.

MAPREDUCE: SIM

Abstract

MapReduce is a programming model and
and generating large datasets that is a
Users specify the computation in terms
lying runtime system automatically parallelizes
machines, handles machine failures, and schedu
cient use of the network and disks. Programme
thousand distinct MapReduce programs have be
past four years, and an average of one hundre
Google's clusters every day, processing a total of

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most the company's products. He is based at Google's Mountain View, CA, facility.

ABSTRACT

We have design
tem, a scalable
data-intensive ap
running on inexp
high aggregate p

While sharing
tributed file syst
vations of our ap
ronment, both cu
departure from
has led us to ree
ically different d

The file syste
It is widely depl
for the generatio
vice as well as re
large data sets. The largest cluster to date provide
dreds of terabytes of storage across thousands of di
over a thousand machines, and it is concurrently ac

ACM Trans. Computer Sys., 26(2): A4, June 2008.

DOI:10.1145/1629175.1629198

MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs.

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for processing and generating large data sets.[4] Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. We built a system around this programming model in 2003 to simplify construction of the inverted index for handling searches at Google.com. Since then, more than 10,000 distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, machine learning, and statistical machine translation. The Hadoop open source implementation of MapReduce has been used extensively outside of Google by a number of organizations.[10,11]

To help illustrate the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code like the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

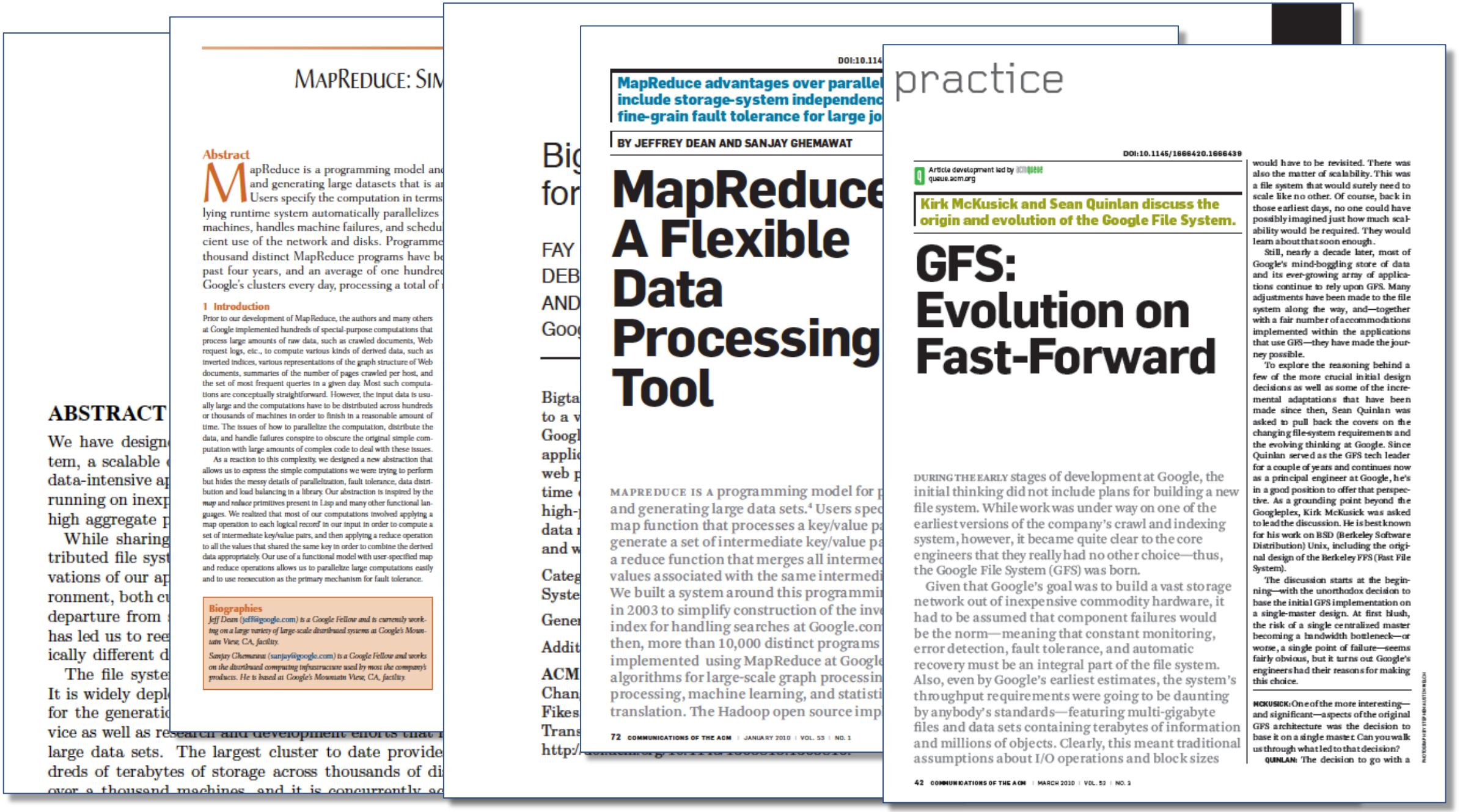
The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing requiredinter-machine communication. MapReduce allows programmers with no experience with parallel and distributed systems to easilyutilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on hundreds or thousands of machines. Programmers find the system easy to use, and more than 100,000 MapReduce jobs are executed on Google's clusters every day.

Compared to Parallel Databases

The query languages built into parallel database systems are also used to

Comms. ACM, 53(1): 72-77, January 2010



practice

DOI:10.1145/1666420.1666439

Article development led by queue.acm.org

Kirk McKusick and Sean Quinlan discuss the origin and evolution of the Google File System.

GFS: Evolution on Fast-Forward

DURING THE EARLY stages of development at Google, the initial thinking did not include plans for building a new file system. While work was under way on one of the earliest versions of the company's crawl and indexing system, however, it became quite clear to the core engineers that they really had no other choice—thus, the Google File System (GFS) was born.

Given that Google's goal was to build a vast storage network out of inexpensive commodity hardware, it had to be assumed that component failures would be the norm—meaning that constant monitoring, error detection, fault tolerance, and automatic recovery must be an integral part of the file system. Also, even by Google's earliest estimates, the system's throughput requirements were going to be daunting by anybody's standards—featuring multi-gigabyte files and data sets containing terabytes of information and millions of objects. Clearly, this meant traditional assumptions about I/O operations and block sizes would have to be revisited. There was also the matter of scalability. This was a file system that would surely need to scale like no other. Of course, back in those earliest days, no one could have possibly imagined just how much scalability would be required. They would learn about that soon enough.

Still, nearly a decade later, most of Google's mind-boggling store of data and its ever-growing array of applications continue to rely upon GFS. Many adjustments have been made to the file system along the way, and—together with a fair number of accommodations implemented within the applications that use GFS—they have made the journey possible.

To explore the reasoning behind a few of the more crucial initial design decisions as well as some of the incremental adaptations that have been made since then, Sean Quinlan was asked to pull back the covers on the changing file-system requirements and the evolving thinking at Google. Since Quinlan served as the GFS tech leader for a couple of years and continues now as a principal engineer at Google, he's in a good position to offer that perspective. As a grounding point beyond the Googleplex, Kirk McKusick was asked to lead the discussion. He is best known for his work on BSD (Berkeley Software Distribution) Unix, including the original design of the Berkeley FFS (Fast File System).

The discussion starts at the beginning—with the unorthodox decision to base the initial GFS implementation on a single-master design. At first blush, the risk of a single centralized master becoming a bandwidth bottleneck—or worse, a single point of failure—seems fairly obvious, but it turns out Google's engineers had their reasons for making this choice.

MCKUSICK: One of the more interesting—and significant—aspects of the original GFS architecture was the decision to base it on a single master. Can you walk us through what led to that decision?

QUINLAN: The decision to go with a

MapReduce advantages over parallel include storage-system independenc fine-grain fault tolerance for large jo

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for p and generating large data sets.[4] Users spec map function that processes a key/value pa generate a set of intermediate key/value pa a reduce function that merges all interme values associated with the same intermedi We built a system around this programmi in 2003 to simplify construction of the inv index for handling searches at Google.com then, more than 10,000 distinct programs implemented using MapReduce at Google algorithms for large-scale graph processin processing, machine learning, and statisti translation. The Hadoop open source imp

MAPREDUCE: SIM

Abstract

MapReduce is a programming model and generating large datasets that is a Users specify the computation in terms lying runtime system automatically parallelizes machines, handles machine failures, and schedu cient use of the network and disks. Programme thousand distinct MapReduce programs have be past four years, and an average of one hundre Google's clusters every day, processing a total of

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most of the company's products. He is based at Google's Mountain View, CA, facility.

ABSTRACT

We have design
tem, a scalable
data-intensive ap
running on inexp
high aggregate p
While sharing
tributed file syst
vations of our ap
ronment, both cu
departure from
has led us to ree
ically different d
The file syste
It is widely depl
for the generatio
vice as well as re
large data sets. The largest cluster to date provide
dreds of terabytes of storage across thousands of di

Big
for

FAY
DEB
AND
Goo

Bigta
to a v
Googl
applic
web p
time
high-
data
and w

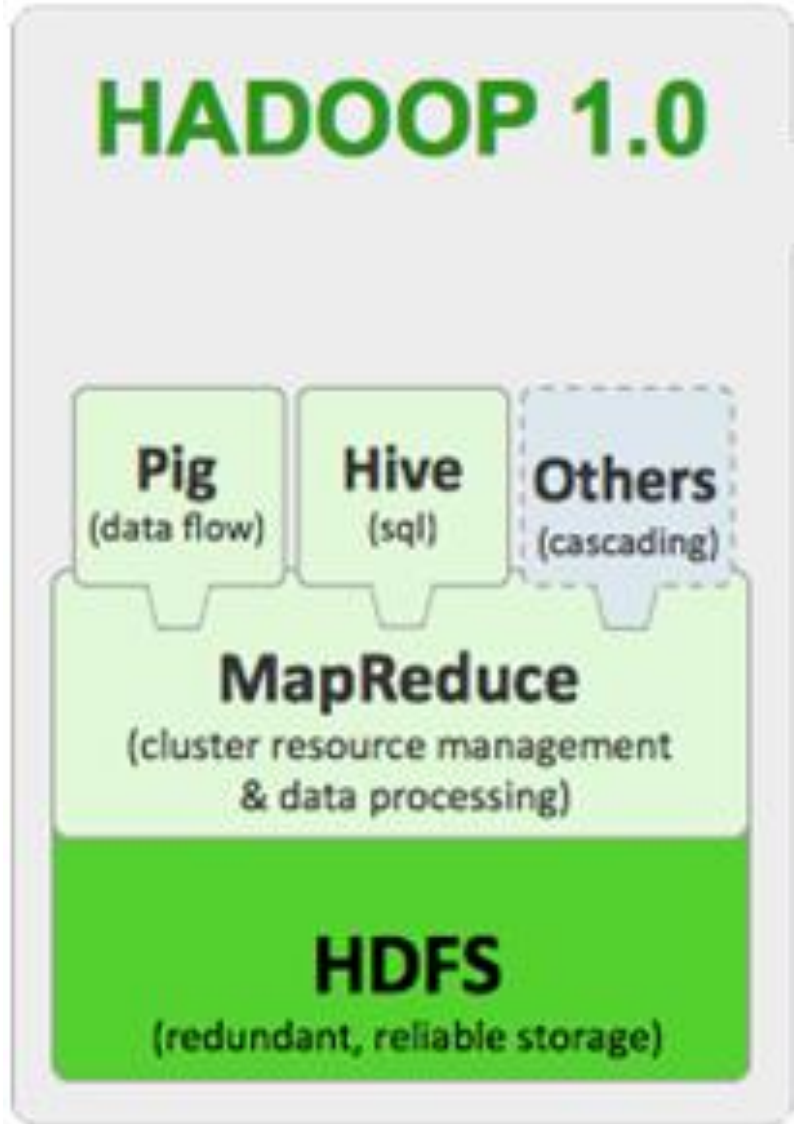
Categ
Syste

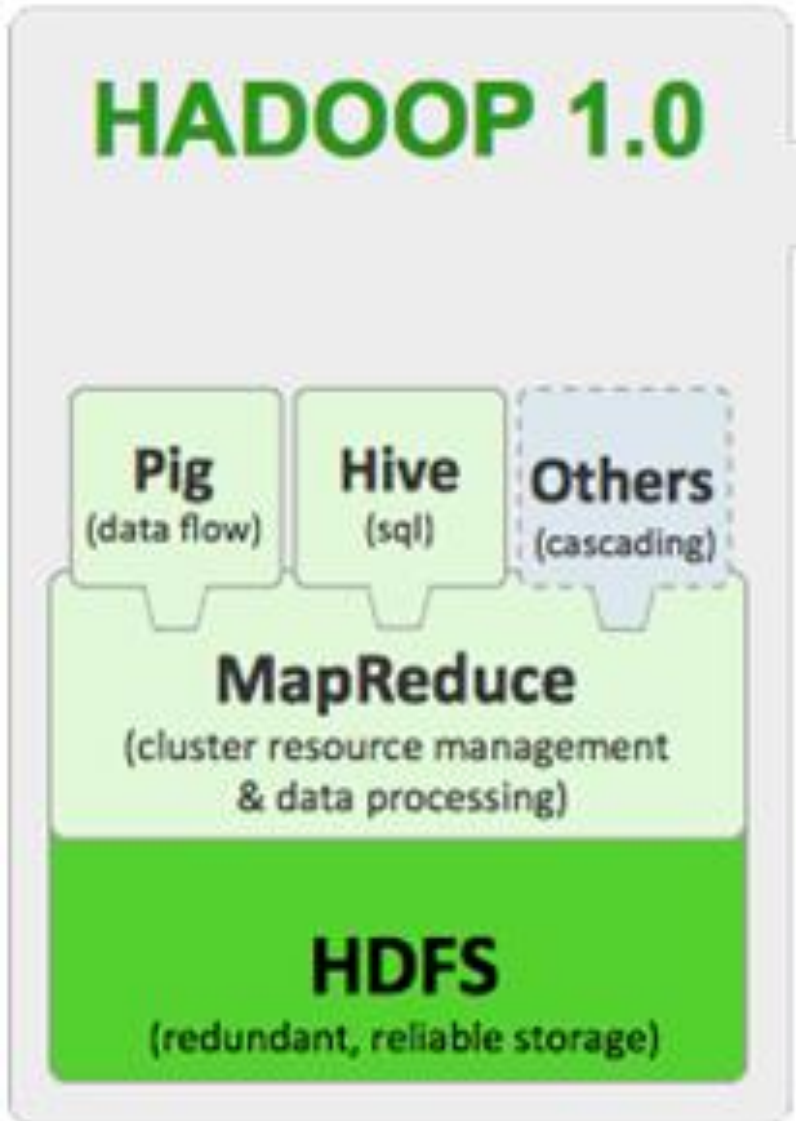
Gener

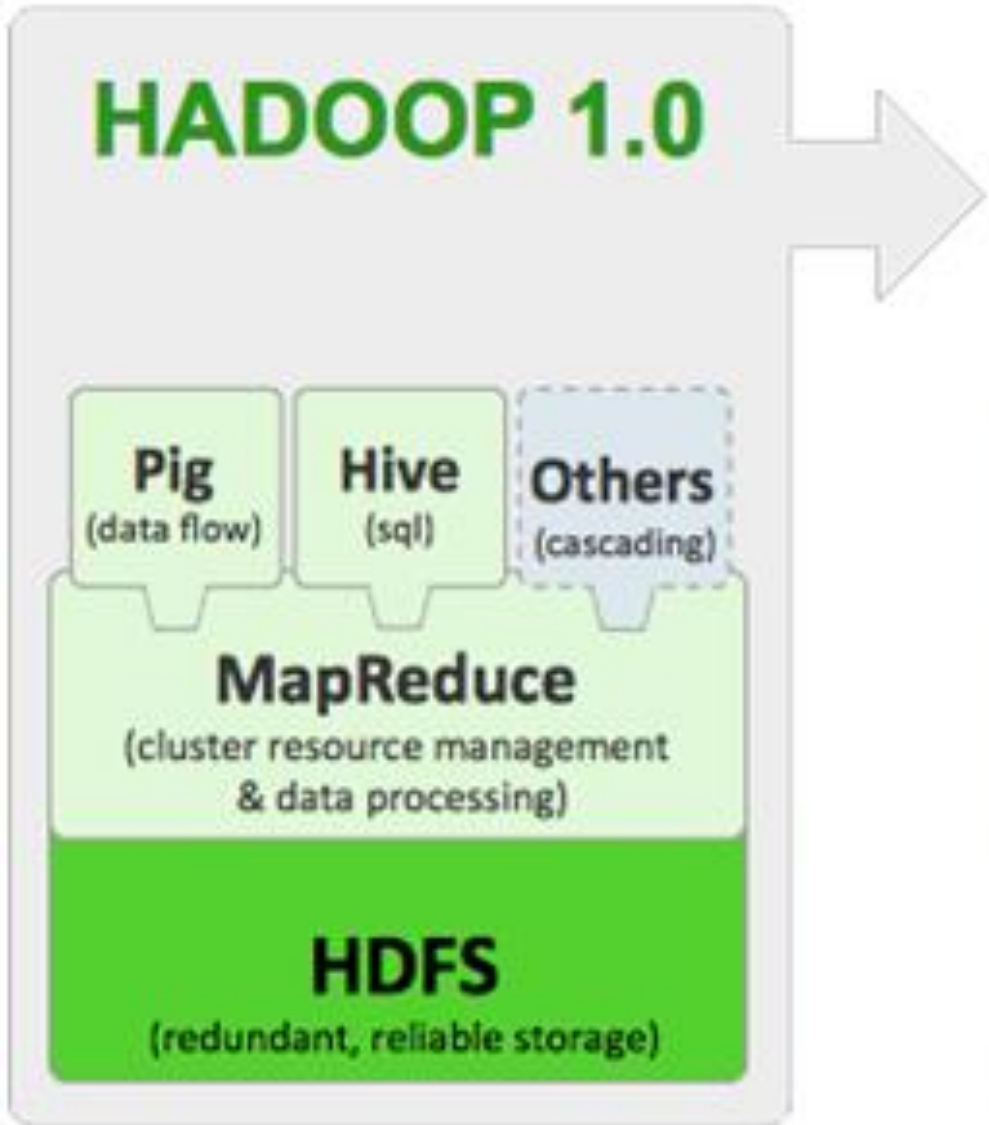
Addit

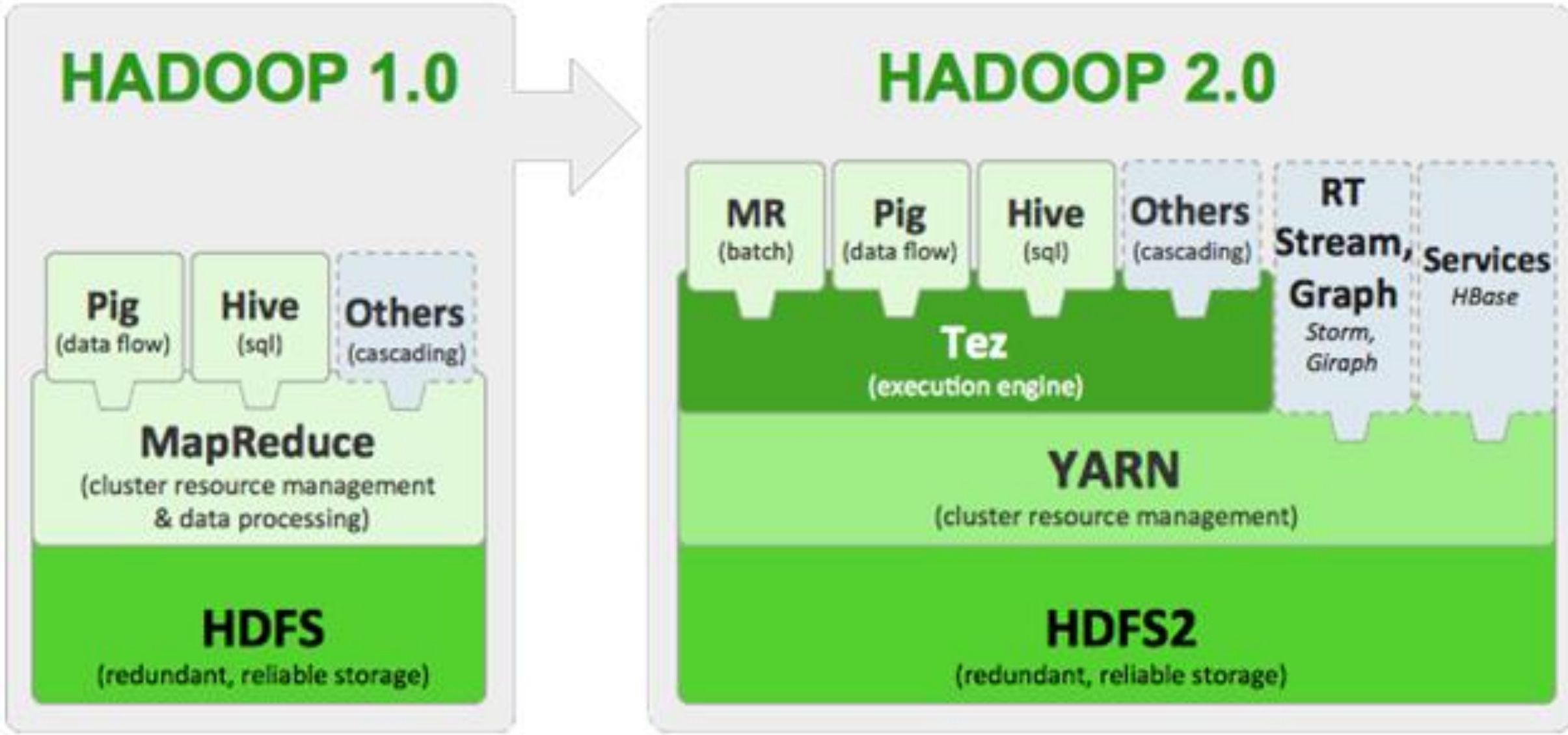
ACM
Chan
Fikes
Trans
http:/

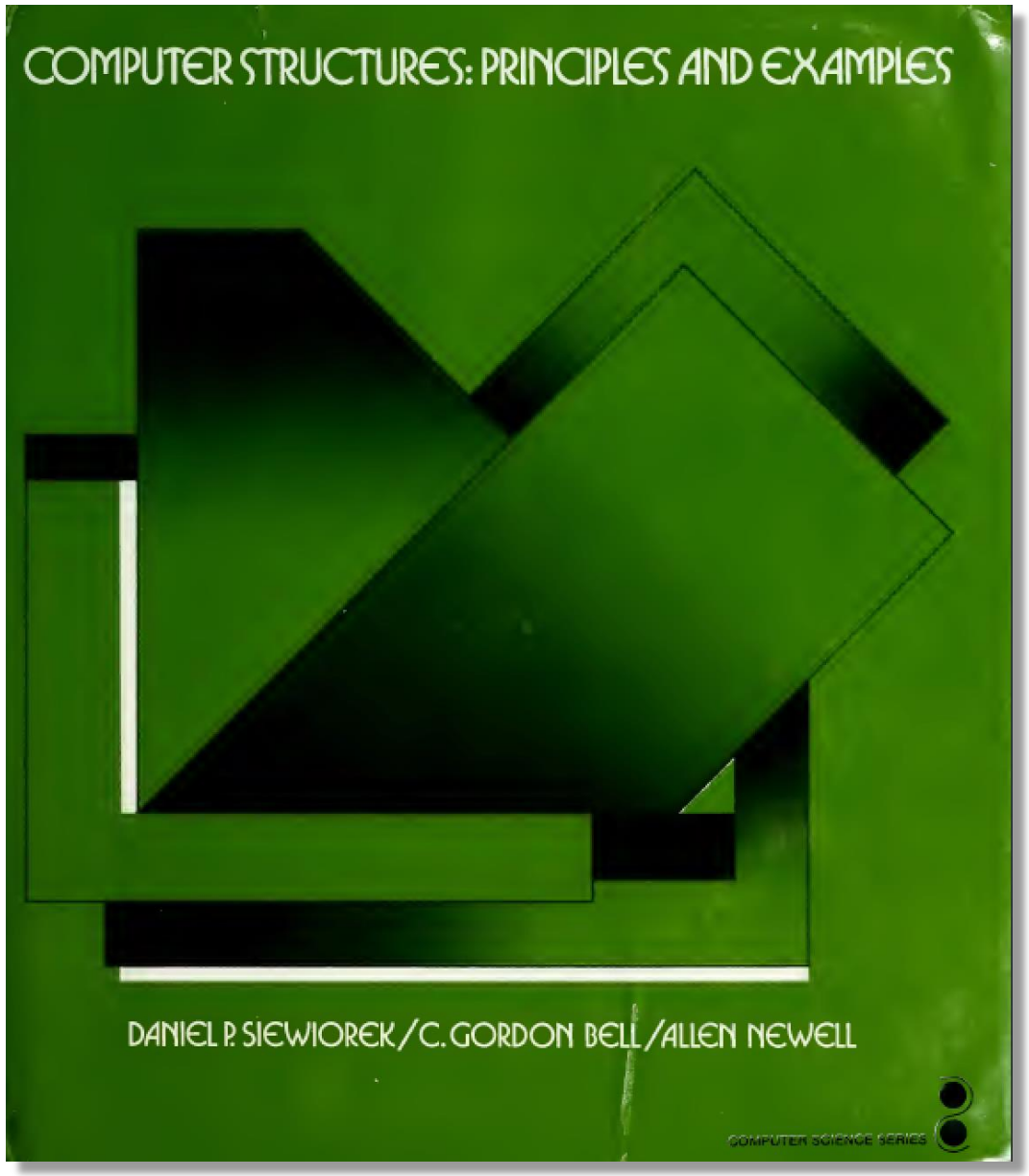
Comm. ACM, 53(3): 42-49, March 2010.


HADOOP 1.0
Pig
(data flow)
Hive
(sql)
Others
(cascading)
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)


HADOOP 1.0
Pig
(data flow)
Hive
(sql)
Others
(cascading)
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)


HADOOP 1.0
Pig
(data flow)
Hive
(sql)
Others
(cascading)
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)


HADOOP 1.0
Pig
(data flow)
Hive
(sql)
Others
(cascading)
MapReduce
(cluster resource management
& data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MR
(batch)
Pig
(data flow)
Hive
(sql)
Others
(cascading)
Tez
(execution engine)
RT
Stream,
Graph
Storm,
Giraph
Services
HBase
YARN
(cluster resource management)
HDFS2
(redundant, reliable storage)


COMPUTER STRUCTURES: PRINCIPLES AND EXAMPLES
DANIEL P. SIEWIOREK/C. GORDON BELL/ALLEN NEWELL
COMPUTER SCIENCE SERIES

1982

Contents in Brief

Preface xiii

Part 1 FUNDAMENTALS

Section 1 ABSTRACTION AND NOTATION 3

CHAPTER 1 Computer Classes and Evolution 4
CHAPTER 2 Levels and Abstractions 9
CHAPTER 3 PMS Notation 17
CHAPTER 4 An Introduction to ISPS 23

Section 2 THE COMPUTER SPACE 33

CHAPTER 5 Function and Performance 39
CHAPTER 6 Structure 62

Section 3 COMPUTERS OF HISTORICAL SIGNIFICANCE 96

CHAPTER 7 The Manchester Mark 1 107
CHAPTER 8 Structural Levels of the PDP-8 110
CHAPTER 9 Design of the B 5000 System 129
CHAPTER 10 One-Level Storage System 135

Part 2 REGIONS OF COMPUTER SPACE

Section 1 MICROPROGRAM-BASED PROCESSORS 151

CHAPTER 11 Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer 158
CHAPTER 12 Microprogramming the IBM System/360 Model 30 164
CHAPTER 13 Bit-Sliced Microprocessor of the Am2900 Family: The Am2901/2909 168
CHAPTER 14 The Am2903/2910 186
CHAPTER 15 A PDP-8 Implemented from AMD Bit-Sliced Microprocessors 219

Section 2 MEMORY HIERARCHIES AND MULTIPLE PROCESSES 227

CHAPTER 16 Burroughs' B6500/B7500 Stack Mechanism 244
CHAPTER 17 An Outline of the ICL 2900 Series System Architecture 251

Section 3 CONCURRENCY: SINGLE-PROCESSOR SYSTEMS 260

CHAPTER 18 The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling 300
CHAPTER 19 An Efficient Algorithm for Exploiting Multiple Arithmetic Units 293
CHAPTER 20 The Illiac IV System 306
CHAPTER 21 A Productive Implementation of an Associative Array Processor: STARAN 317

Section 4 MULTIPLE-PROCESSOR SYSTEMS 332

CHAPTER 22 The C.mmp/Hydra Project: An Architectural Overview 350
CHAPTER 23 Pluribus: An Operational Fault-Tolerant Multiprocessor 371

Section 5 NETWORKS 387

CHAPTER 24 The Interface Message Processor for the ARPA Computer Network 402
CHAPTER 25 ALOHA Packet Broadcasting: A Retrospect 416
CHAPTER 26 Ethernet: Distributed Packet Switching for Local Computer Networks 429

Section 6 FAULT-TOLERANT SYSTEMS 439

CHAPTER 27 The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design 448
CHAPTER 28 Fault-Tolerant Design of Local ESS Processors 459
CHAPTER 29 The Tandem 16: A Fault-Tolerant Computing System 470

Section 7 LANGUAGE-BASED COMPUTERS 486

CHAPTER 30 The SYMBOL Computer 489
CHAPTER 31 A Dual-Processor Desk-Top Computer: The HP 9845A 508
CHAPTER 32 The IBM System/38 533

Section 8 PERSONAL COMPUTING SYSTEMS 547

CHAPTER 33 Alto: A Personal Computer 549

Part 3 COMPUTER CLASSES

Section 1 MONOLITHIC MICROCOMPUTERS 581

CHAPTER 34 TMS1000/1200: Chip Architecture and Operation 587
CHAPTER 35 PIC1650: Chip Architecture and Operation 602

Section 2 MICROCOMPUTERS 610

CHAPTER 36 Trends in Microcomputers 612
CHAPTER 37 Intel Microprocessors: 8008 to 8086 615

Section 3 MINICOMPUTERS 647

CHAPTER 38 A New Architecture for Mini-Computers: The DEC PDP-11 649
CHAPTER 39 Implementation and Performance Evaluation of the PDP-11 Family 666

Section 4 MAXICOMPUTERS 680

CHAPTER 40 The Structure of System/360, Part I: Outline of the Logical Structure 695
CHAPTER 41 The Structure of System/360, Part II: System Implementations 711
CHAPTER 42 VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family 716
CHAPTER 43 Parallel Operation in the Control Data 6600 730
CHAPTER 44 The CRAY-1 Computer System 743
CHAPTER 45 The TI ASC: A Highly Modular and Flexible Super Computer Architecture 753

Part 4 FAMILY RANGE, COMPATIBILITY, AND EVOLUTION

Section 1 MICROCOMPUTER FAMILIES 765

Section 2 MINICOMPUTER FAMILIES 766

CHAPTER 46 The PDP-8 Family 767
CHAPTER 47 The Evolution of the PDP-11 776

Section 3 EVOLUTION OF HP CALCULATORS 785

CHAPTER 48 The HP Model 9100A Computing Calculator 787
CHAPTER 49 The HP 9810/20/30 Series 801
CHAPTER 50 Hewlett-Packard Calculator Architectures 824

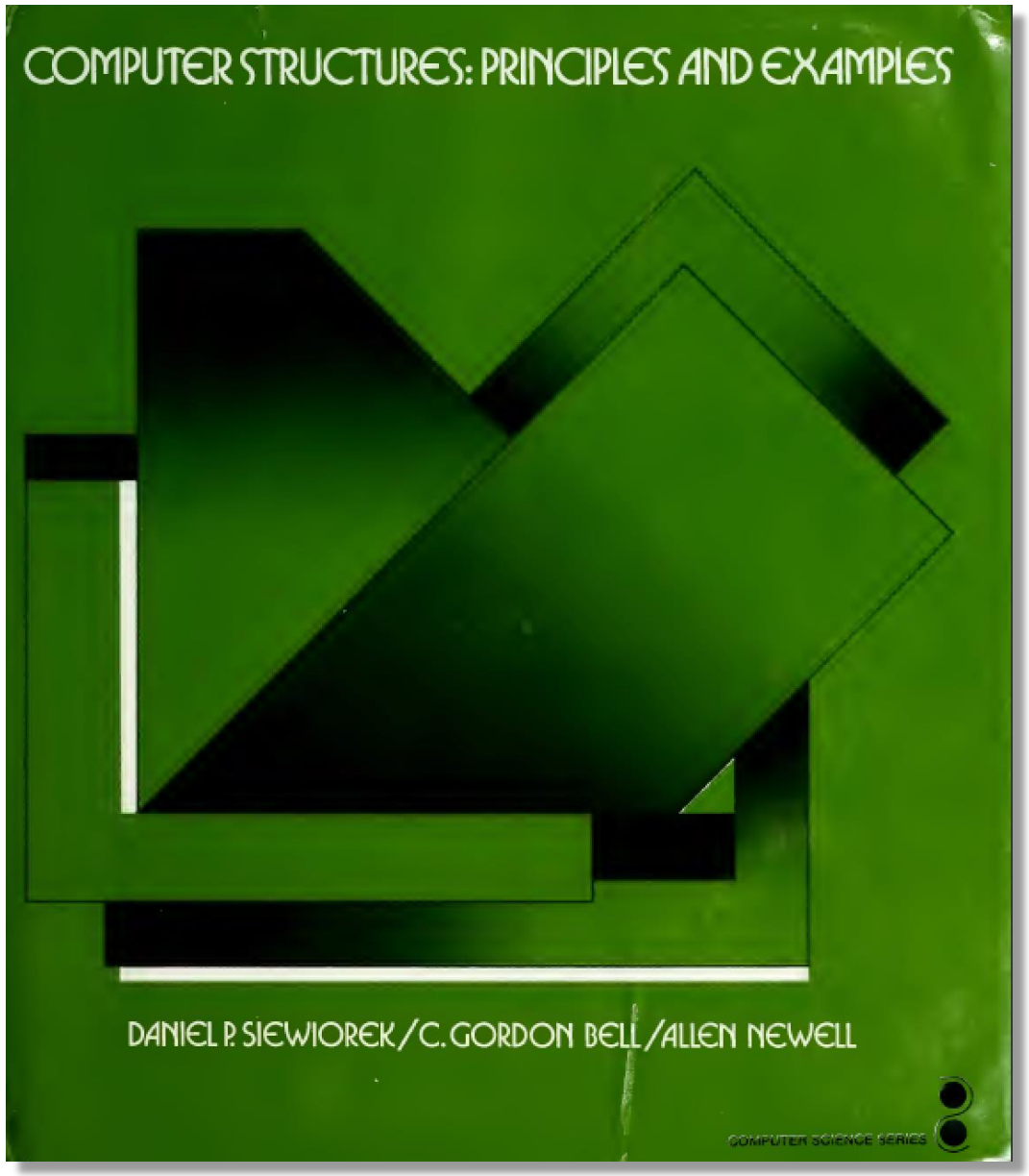
Section 4 EVOLUTION OF BURROUGHS COMPUTERS 828

Section 5 THE SYSTEM/360 AND SYSTEM/370 FAMILY 829

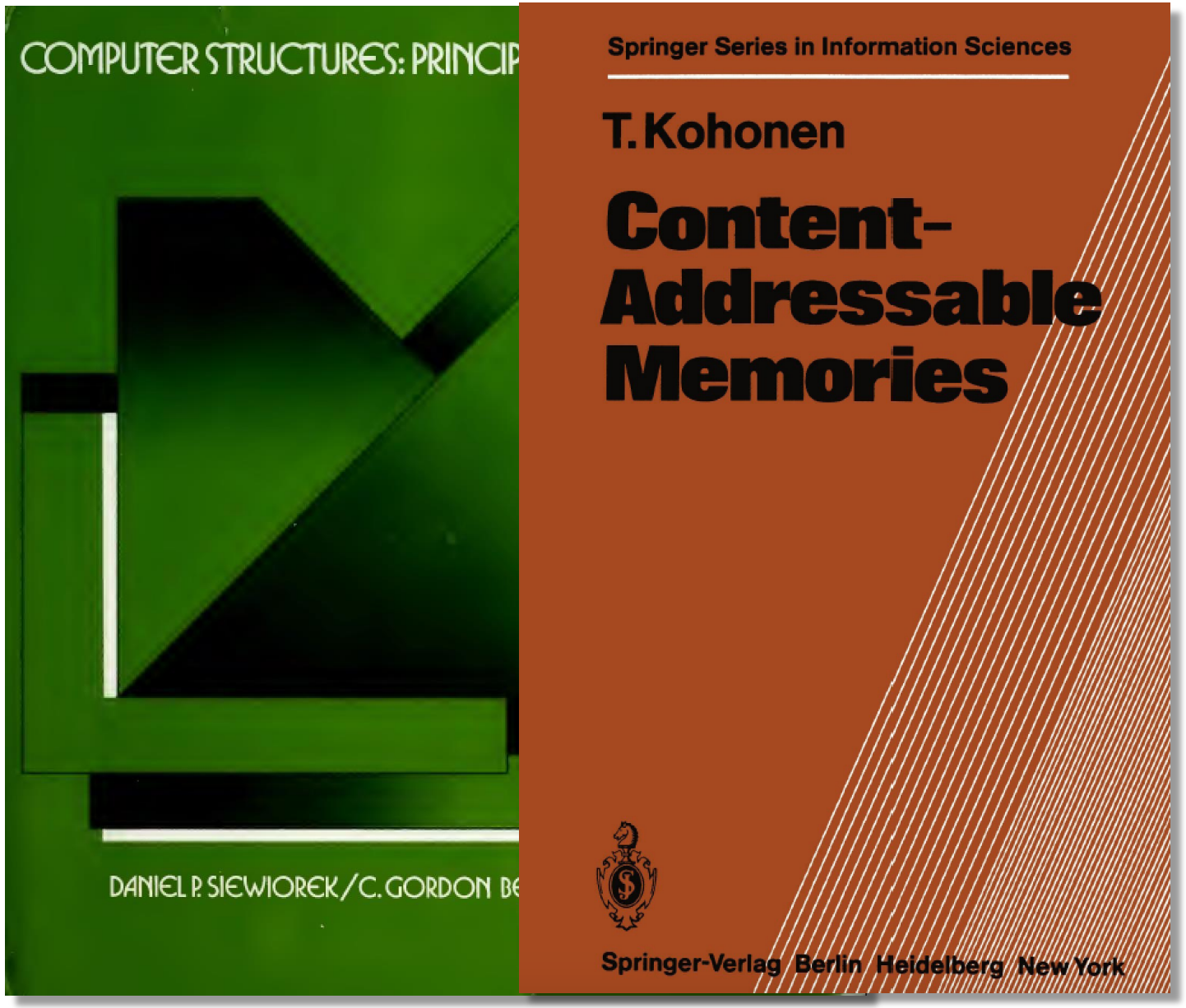
CHAPTER 51 Architecture of the IBM System/370 830
CHAPTER 52 The IBM System/360, System/370, 3030, and 4300: A Series of Planned Machines That Span a Wide Performance Range 856

Section 6 EVOLUTION OF CDC/CRAY COMPUTERS 893

Bibliography 894
Acknowledgments 915
Index 921

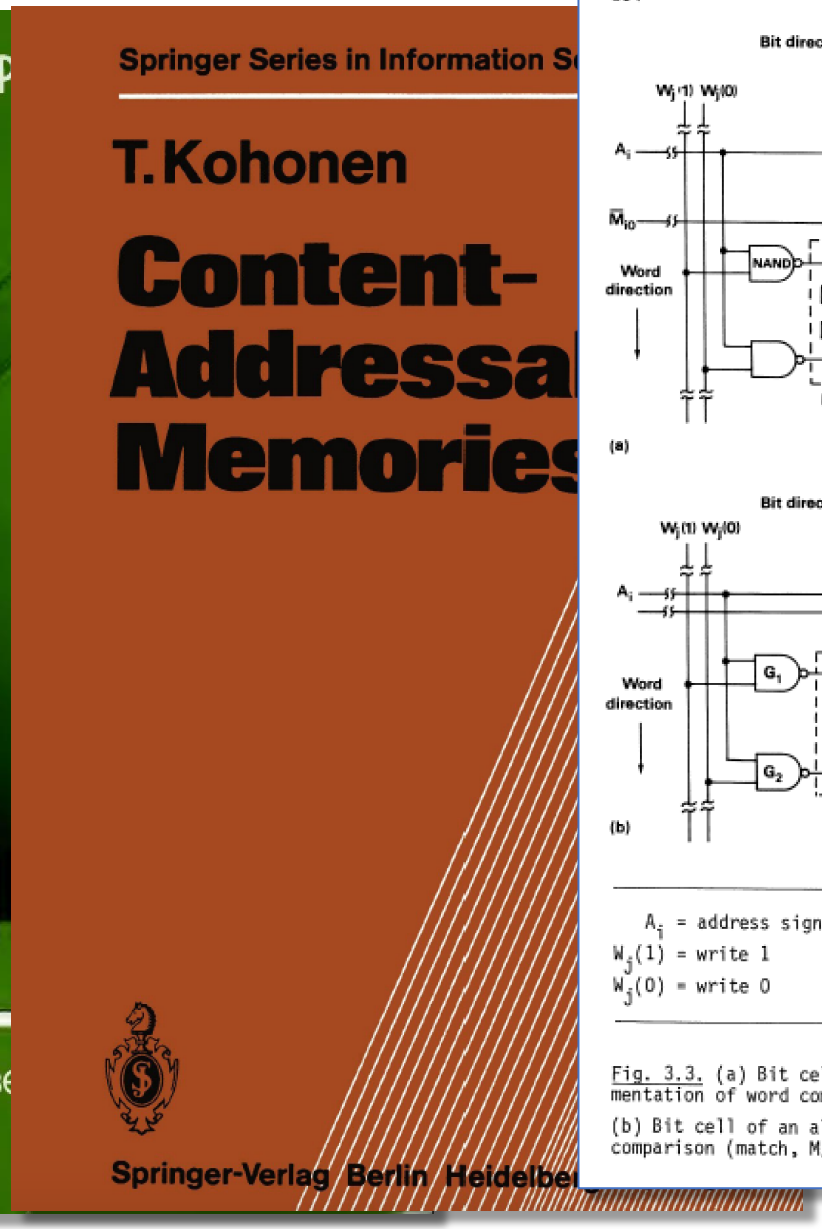

COMPUTER STRUCTURES: PRINCIPLES AND EXAMPLES
DANIEL P. SIEWIOREK/C. GORDON BELL/ALLEN NEWELL
COMPUTER SCIENCE SERIES

1982


COMPUTER STRUCTURES: PRINCIP
DANIEL P. SIEWIOREK/C. GORDON BE
Springer Series in Information Sciences
T. Kohonen
Content-Addressable Memories
Springer-Verlag Berlin Heidelberg New York

1982 1980

1982



1980

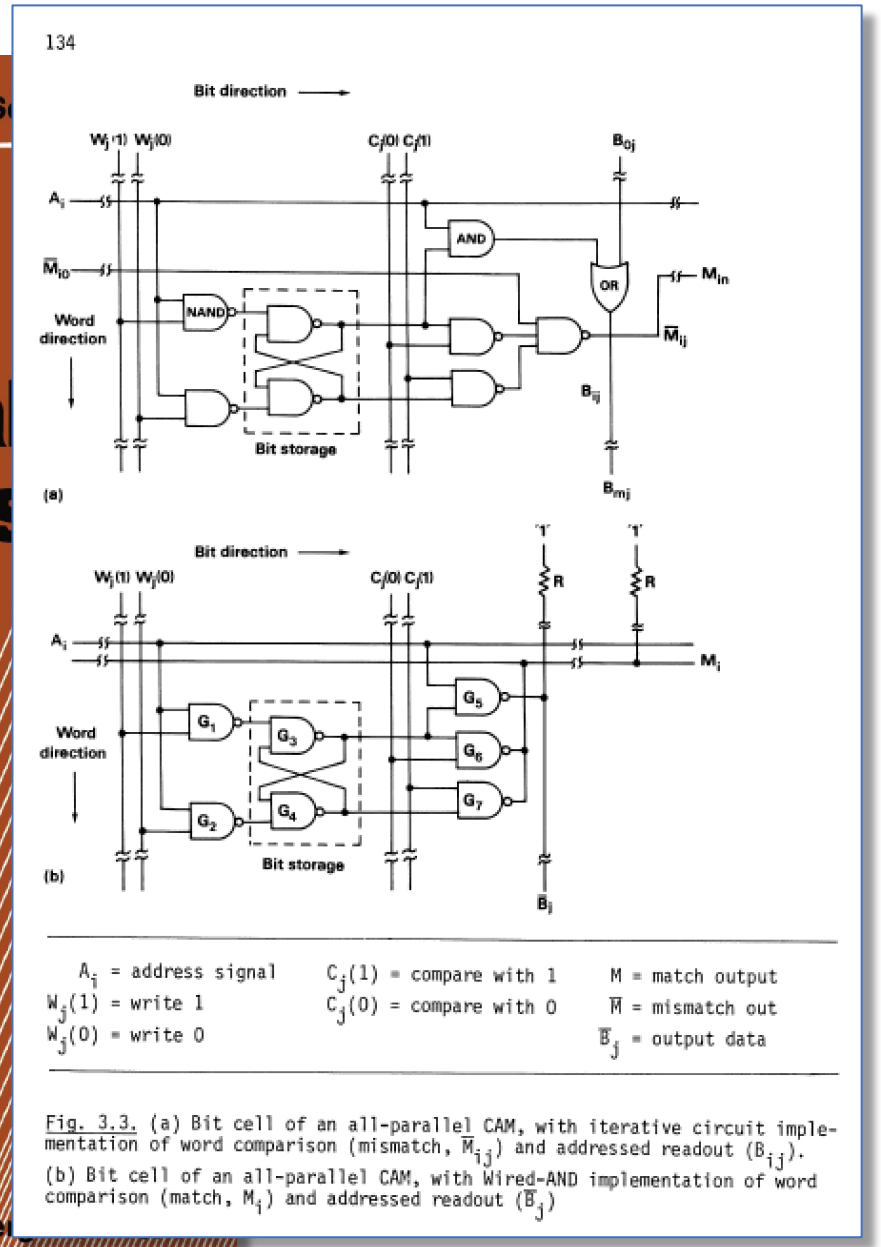


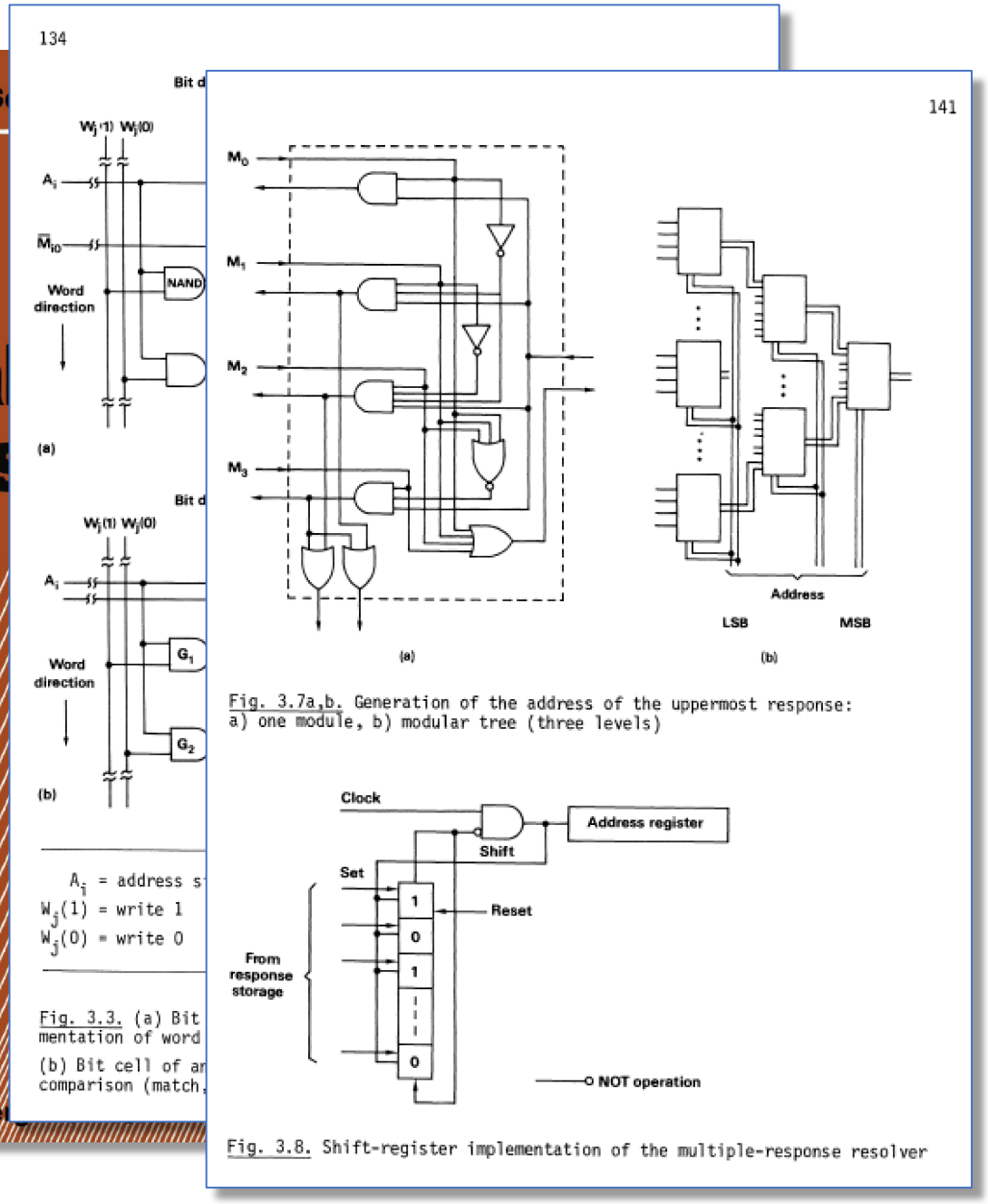
Fig. 3.3. (a) Bit cell of an all-parallel CAM, with iterative circuit implementation of word comparison (mismatch, $\overline{M}_{ij}$) and addressed readout (B_{ij}). (b) Bit cell of an all-parallel CAM, with Wired-AND implementation of word comparison (match, M_i) and addressed readout ($\overline{B}_j$)

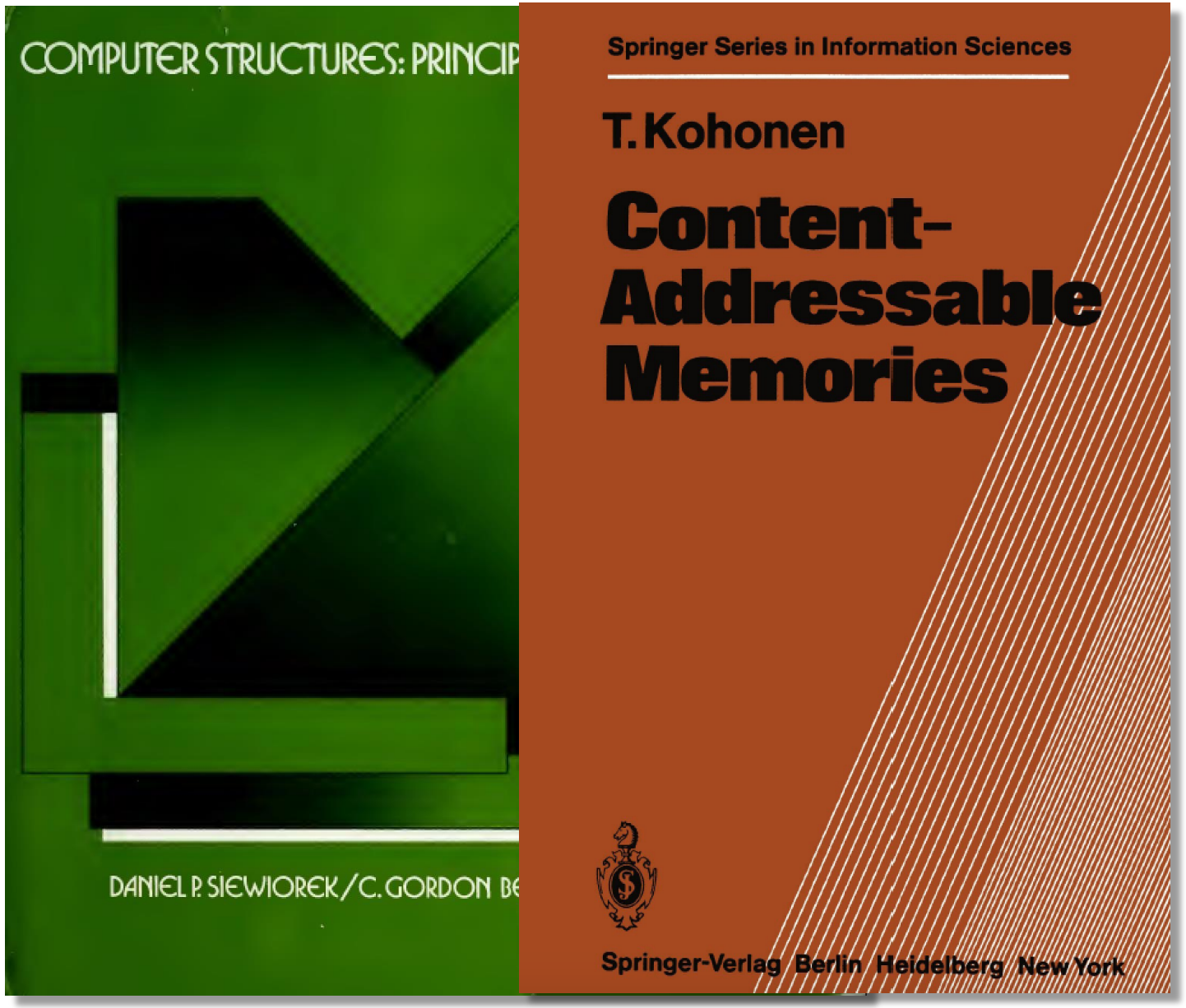
COMPUTER STRUCTURES: PRINCIP
DANIEL P. SIEWIOREK/C. GORDON BE

1982

Springer Series in Information Sc
T. Kohonen
Content-
Addressa
Memories
Springer-Verlag Berlin Heidelbe

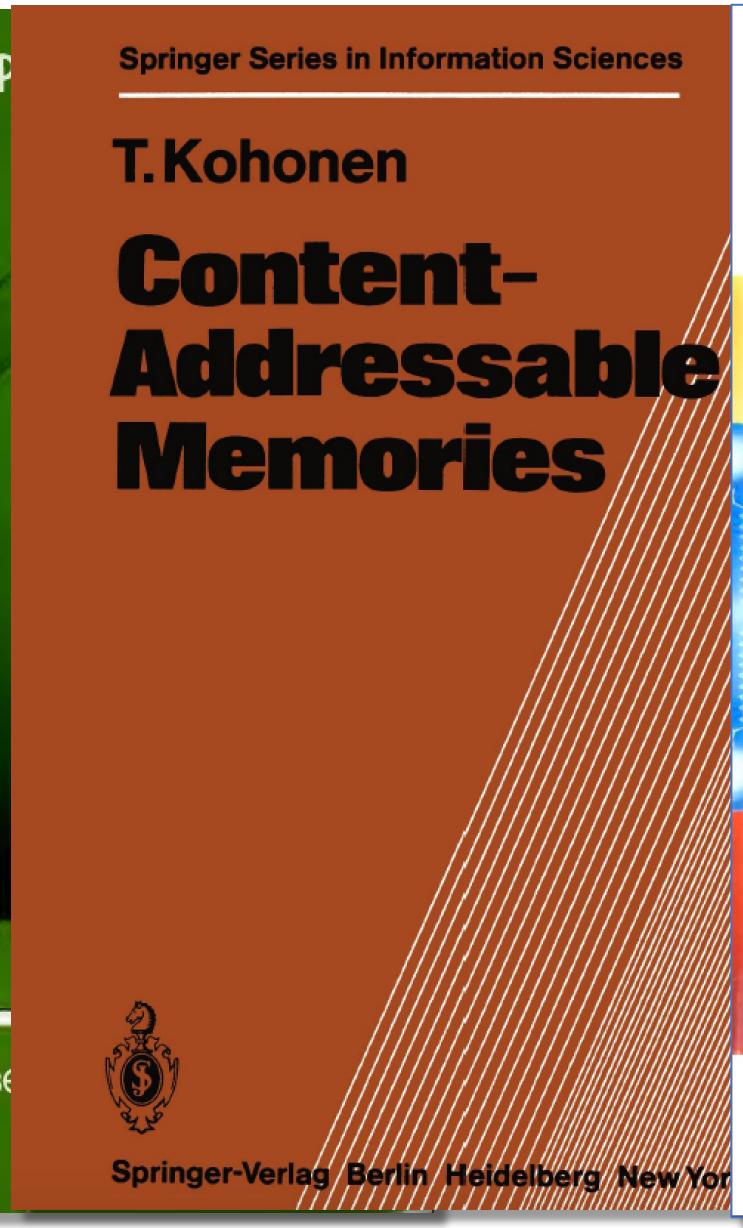
1980


134
Bit d
Word direction
NAND
(a)
Bit d
Word direction
(b)
A_i = address s
W_j(1) = write 1
W_j(0) = write 0
Fig. 3.3. (a) Bit
mentation of word
(b) Bit cell of ar
comparison (match,
141
M0
M1
M2
M3
Address
LSB
MSB
(a)
(b)
Fig. 3.7a,b. Generation of the address of the uppermost response:
a) one module, b) modular tree (three levels)
Clock
Address register
Shift
Set
Reset
From response storage
1
0
1
0
NOT operation
Fig. 3.8. Shift-register implementation of the multiple-response resolver

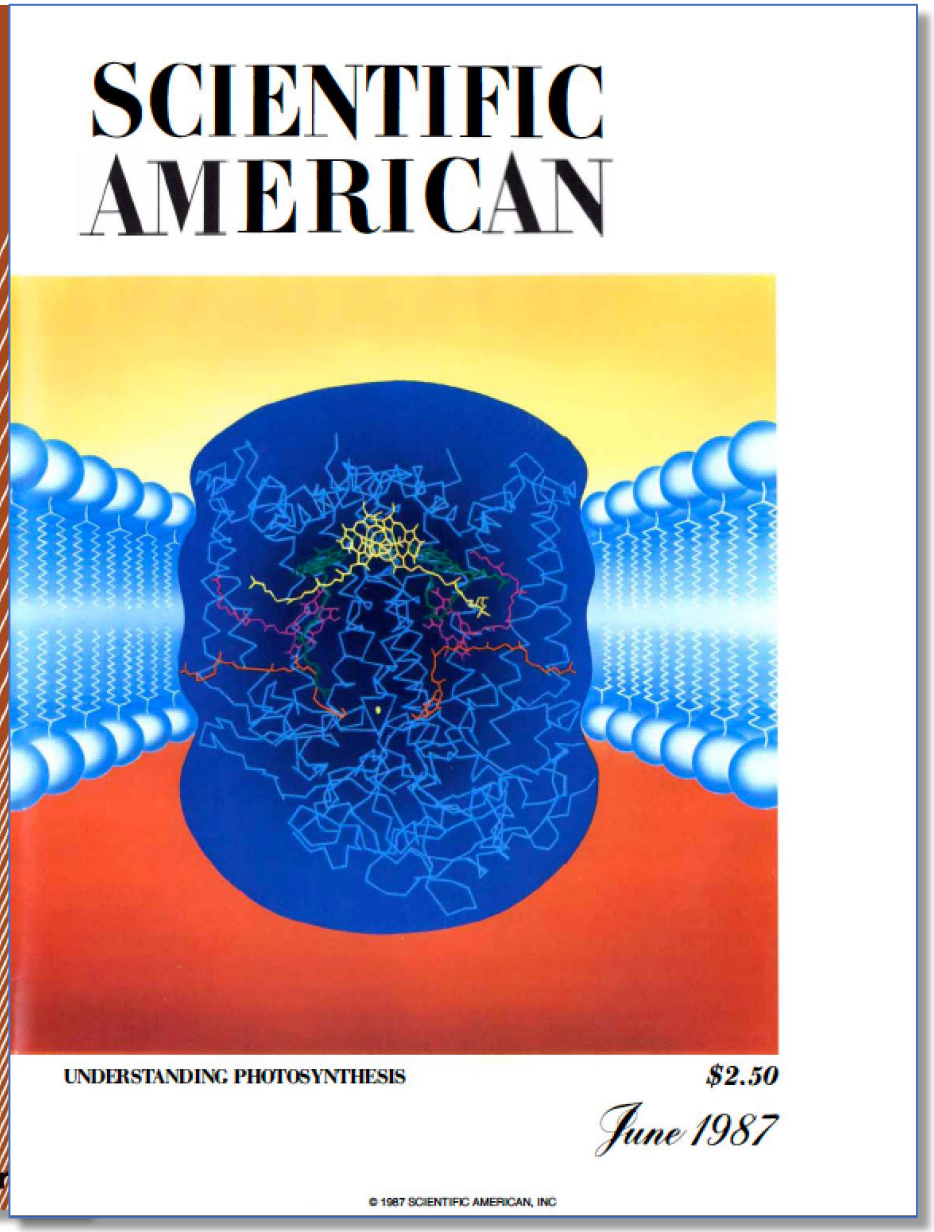

COMPUTER STRUCTURES: PRINCIP
DANIEL P. SIEWIOREK/C. GORDON BE
Springer Series in Information Sciences
T. Kohonen
Content-Addressable Memories
Springer-Verlag Berlin Heidelberg New York

1982 1980

1982



1980



1987

The Connection Machine

Most computers have a single processing unit. In this new parallel computer 65,536 processors work on a problem at once. The resulting speed may transform several fields, including artificial intelligence

by W. Daniel Hillis

In the past three decades remarkable changes have taken place in digital computers. The amount of computational power that once required a room full of vacuum tubes can now be found in hand-held devices. Complex computations that would once have taken days to perform can now be done in seconds. Yet in certain fundamental respects the design of the digital computer remained unchanged between the days of the ENIAC (one of the first large-scale digital machines, built at the University of Pennsylvania in the late 1940's) and the current generation of supercomputers. Most modern computers—from supercomputers to microprocessors—are similar to the ENIAC in that the memory and the central processing unit are separate entities. For a computation to be performed, the appropriate data must be retrieved from memory and brought to the central processor; there it is operated on before being returned to memory.

Such a design is called sequential because the processing operations are performed one at a time. The sequential design was adopted mainly for utilitarian reasons. In the early days of digital computing the memory and the central processing unit were made of different materials. Since memory was cheaper than processing, it was desirable to maximize the efficiency of the processing unit at the expense of the memory's efficiency. And that is just what the sequential design does. Today, however, the memory and the central processor are fabricated from the same etched silicon wafers. In a typical computer more than 90 percent of the silicon is devoted to memory. While the central processor is kept wonderfully busy, this vast majority largely sits idle. At about $1 million per square meter, processed packaged silicon is an expensive resource to waste.

Clearly, the general solution to this problem is to find a way to unify processing capacity and memory. But how? One answer is to exploit many small processors, working simultaneously, each accompanied by a small memory of its own. In such a design, which is called parallel processing, memory capacity and processing capacity can both be utilized with high efficiency. This is the approach my colleagues and I have taken in building a parallel computer called the Connection Machine. The Connection Machine incorporates 65,536 simple processors. Each processor is much less powerful than a typical personal computer, but in tandem they can execute several billion instructions per second, a rate that makes the Connection Machine one of the fastest computers ever constructed.

Yet the most interesting thing about the Connection Machine is not its brute speed but its flexibility. Special-purpose devices have been built that exploit parallelism to perform specific tasks quite quickly. Like idiot savants, however, such machines are usually quite awkward outside their specialties. In contrast, the Connection Machine can operate at its peak processing rate in a wide range of applications. As this article will describe, the key to such flexibility is a communications network that enables the multitude of processors to exchange information in the pattern best suited to the problem at hand. The Connection Machine is not just a prototype. About a dozen Connection Machines are already in commercial use, and they have begun to change the way digital computing treats problems in physics, image processing, text retrieval and even artificial intelligence.

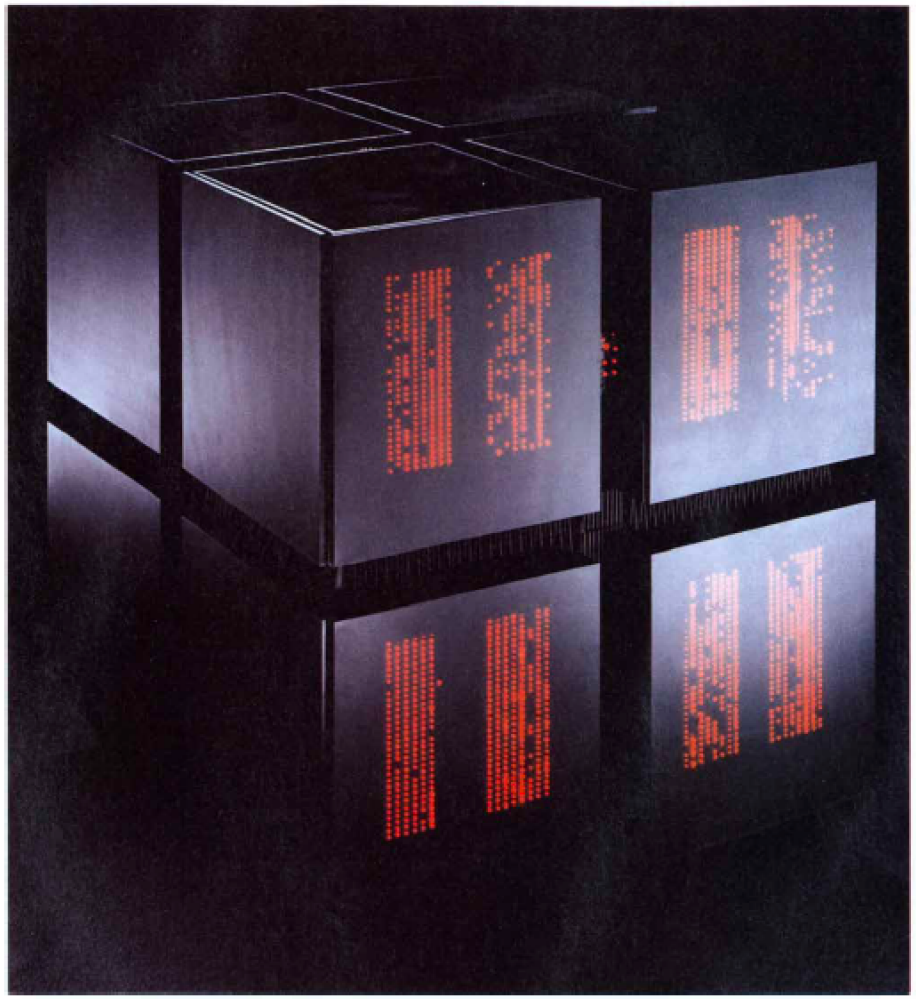
In order to understand the benefits of parallel processing, it is helpful to think about the difference between the way a conventional computer deals with an image and the way the same image is treated in the human brain. From the pair of two-dimensional images falling on the retinas a human being is able—without apparent effort—to reconstruct a three-dimensional model of the world and maintain that model as the two-dimensional images change rapidly. Computers can be programmed to carry out part of the task, but even quite fast computers take hours to do what the human brain can do in fractions of a second [see "Vision by Man and Machine," by Tomaso Poggio; SCIENTIFIC AMERICAN, April, 1984]. The brain maintains its advantage in spite of the fact that its components—neurons—are apparently millions of times slower than the computer's transistors.

Why, then, is the brain so much faster than the computer? The visual circuitry of the brain is not fully understood, but it is clear that in some areas of the brain the principles of parallel processing are at work. In those parts of the brain the entire image is processed at once. The computer, however, examines the image one tiny spot at a time, as if it were looking through a minute keyhole. In the computer the image is represented as an array of numbers, each of which corresponds to the intensity of the light at a particular point. A typical low-resolution array might be a square with 256 points on a side. A conventional computer operates on only one of the square's 65,536 points at a time. Hence even a simple image-processing operation includes 65,536 steps.

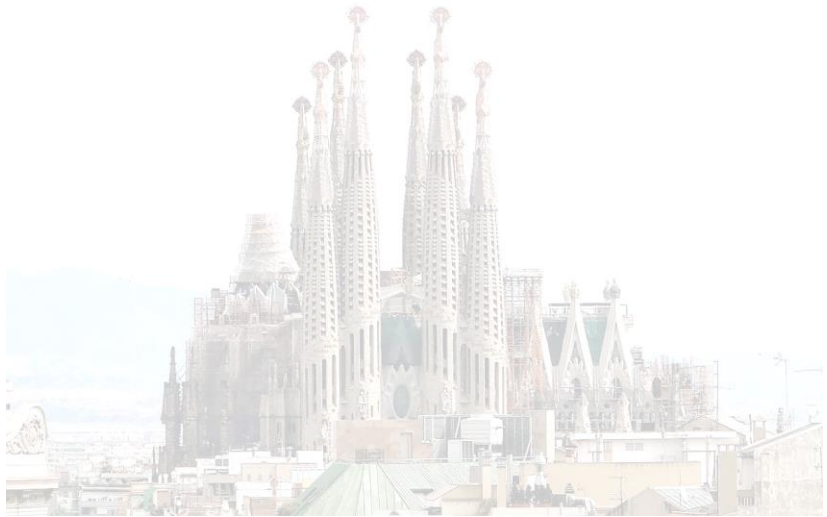
The Connection Machine, on the other hand, assigns a single processor to each point of the image. Since every operation can be performed on all the points simultaneously, a calculation involving the entire image is as fast as a calculation involving only a single point. For example, to find all the points in the image that are brighter than a certain minimum a sequential machine must check the 65,536 tiny elements in succession, comparing each one with the threshold value. In the Connection Machine that comparison is made simultaneously by the 65,536 processors—each one operating on a single element of the image.

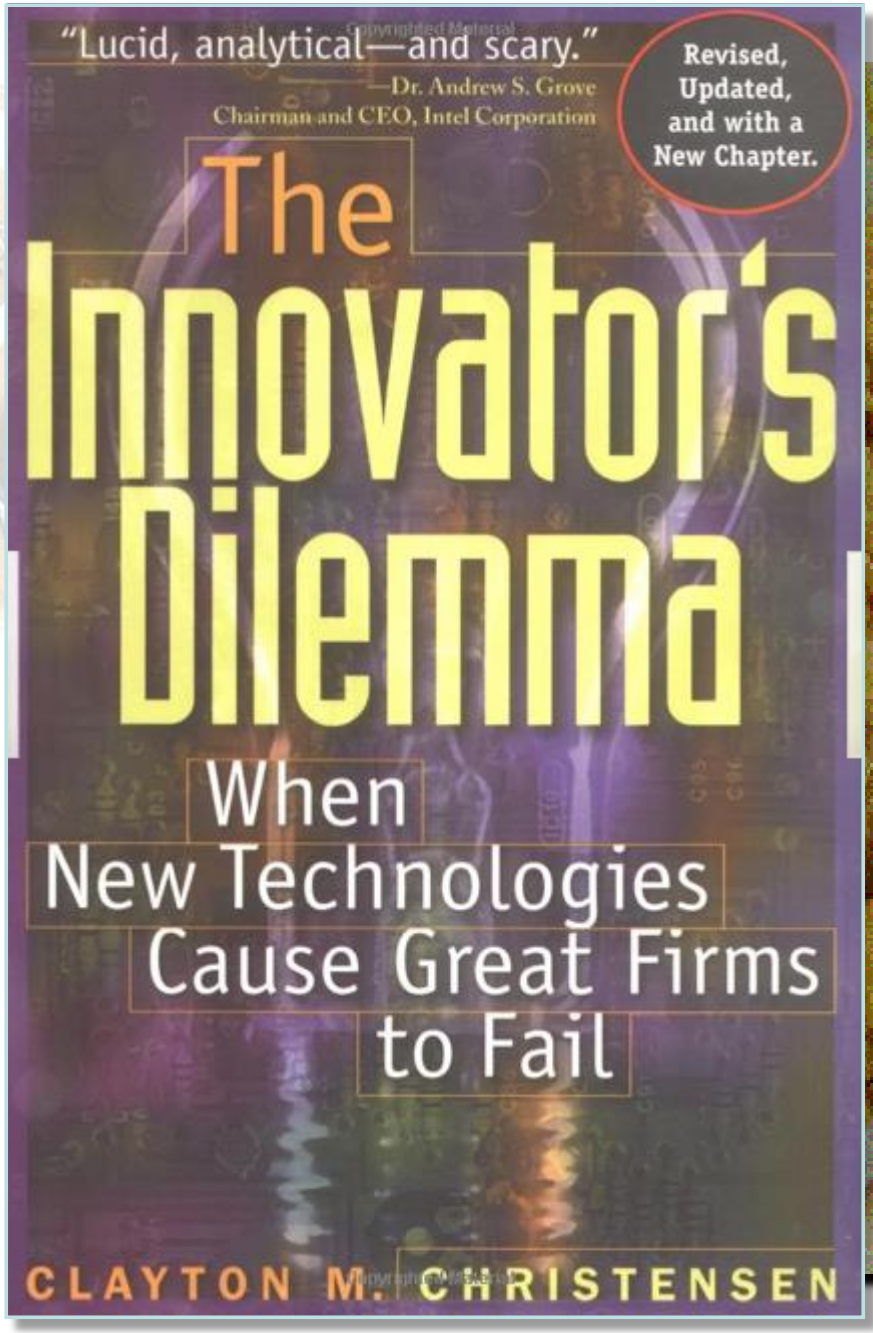
The threshold comparison is particularly simple because it can be carried out independently by each processor. Most interesting computations, however, require that the processors exchange information as the operation proceeds. Consider the common image-processing operation called convolution. Convolution blurs an image by averaging each point with its nearest neighbors in the two-dimensional grid. (Convolution, which is analogous to operations carried out in the human visual system, is useful for removing insignificant details and bringing out significant objects.)

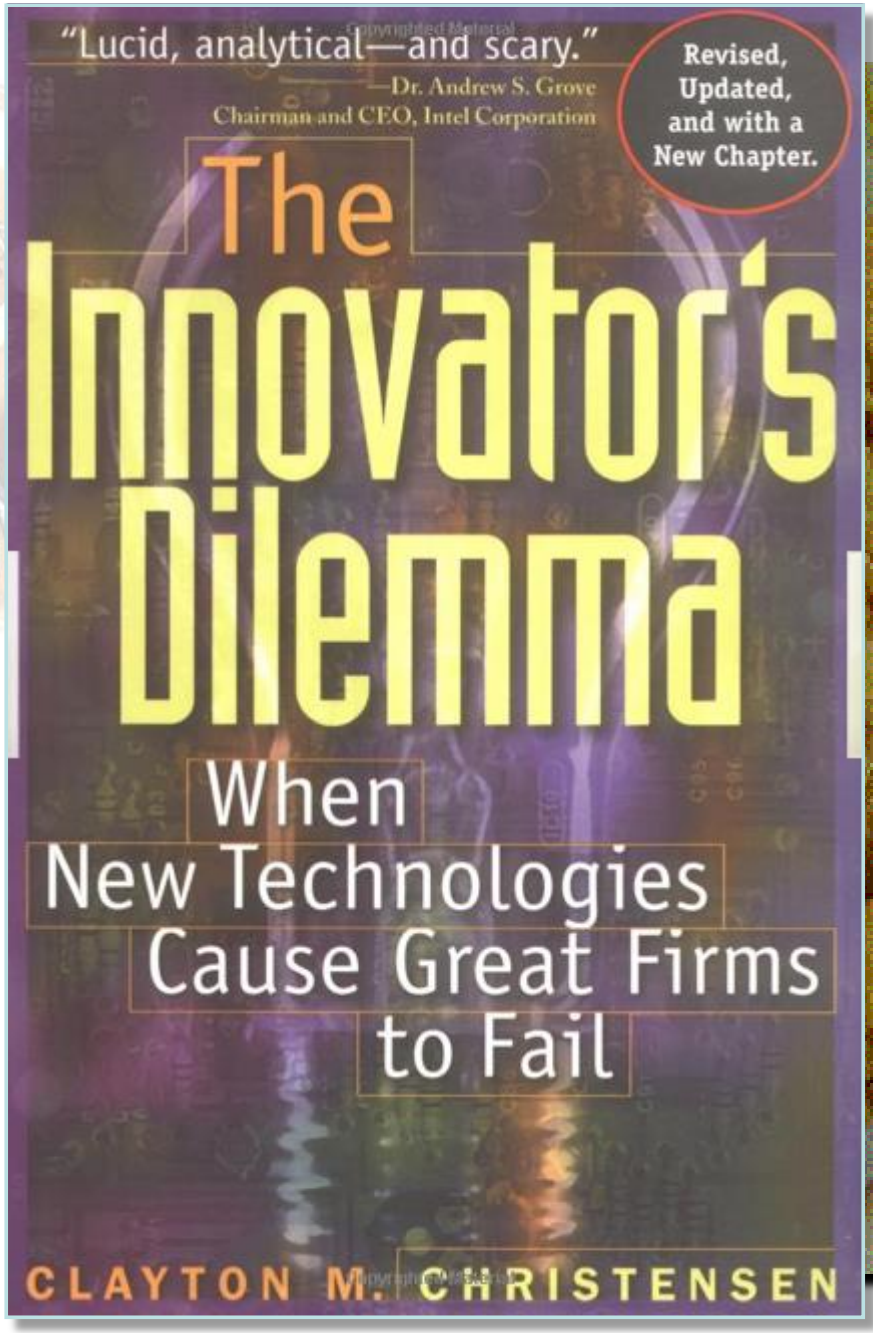
To complete a convolution each processor must read a value from the processors that store information about the points to the left, right, above and below the point in question.

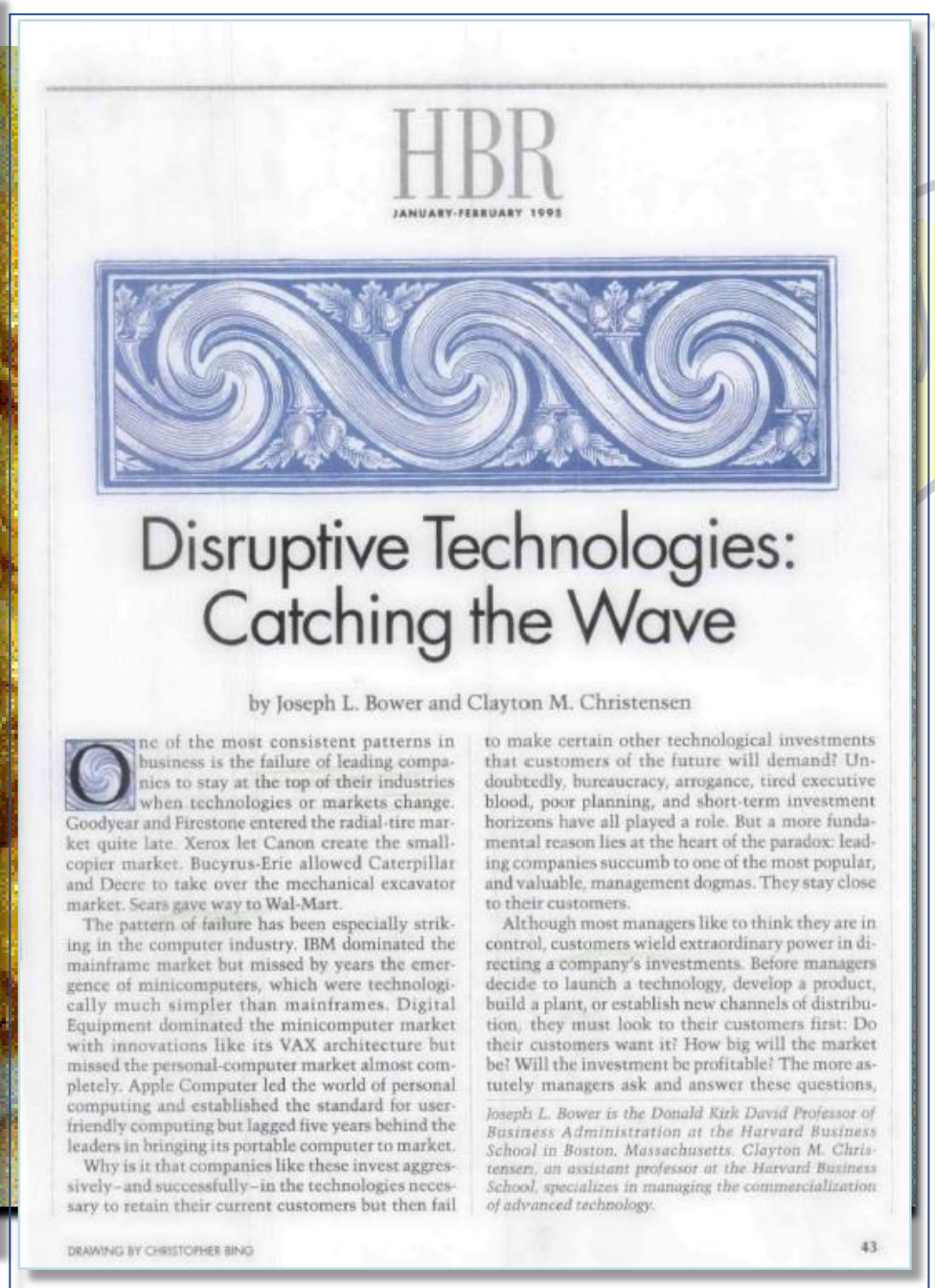


CONNECTION MACHINE is a cube 1.5 meters on a side made up of eight subcubes. Each subcube contains 16 boards arranged vertically. On each board are 32 custom chips. Every chip includes 16 processors, each with a small amount of associated memory. The red lights on the boards indicate the status of the chips; they are for troubleshooting. Operating in parallel, the 65,536 processors can execute several billion instructions per second, making the Connection Machine one of the fastest computers ever built.




"Lucid, analytical—and scary."
—Dr. Andrew S. Grove
Chairman and CEO, Intel Corporation
Revised, Updated, and with a New Chapter.
The Innovator's Dilemma
When New Technologies Cause Great Firms to Fail
CLAYTON M. CHRISTENSEN





HBR

JANUARY-FEBRUARY 1995

Disruptive Technologies: Catching the Wave

by Joseph L. Bower and Clayton M. Christensen

One of the most consistent patterns in business is the failure of leading companies to stay at the top of their industries when technologies or markets change. Goodyear and Firestone entered the radial-tire market quite late. Xerox let Canon create the small-copier market. Bucyrus-Erie allowed Caterpillar and Deere to take over the mechanical excavator market. Sears gave way to Wal-Mart.

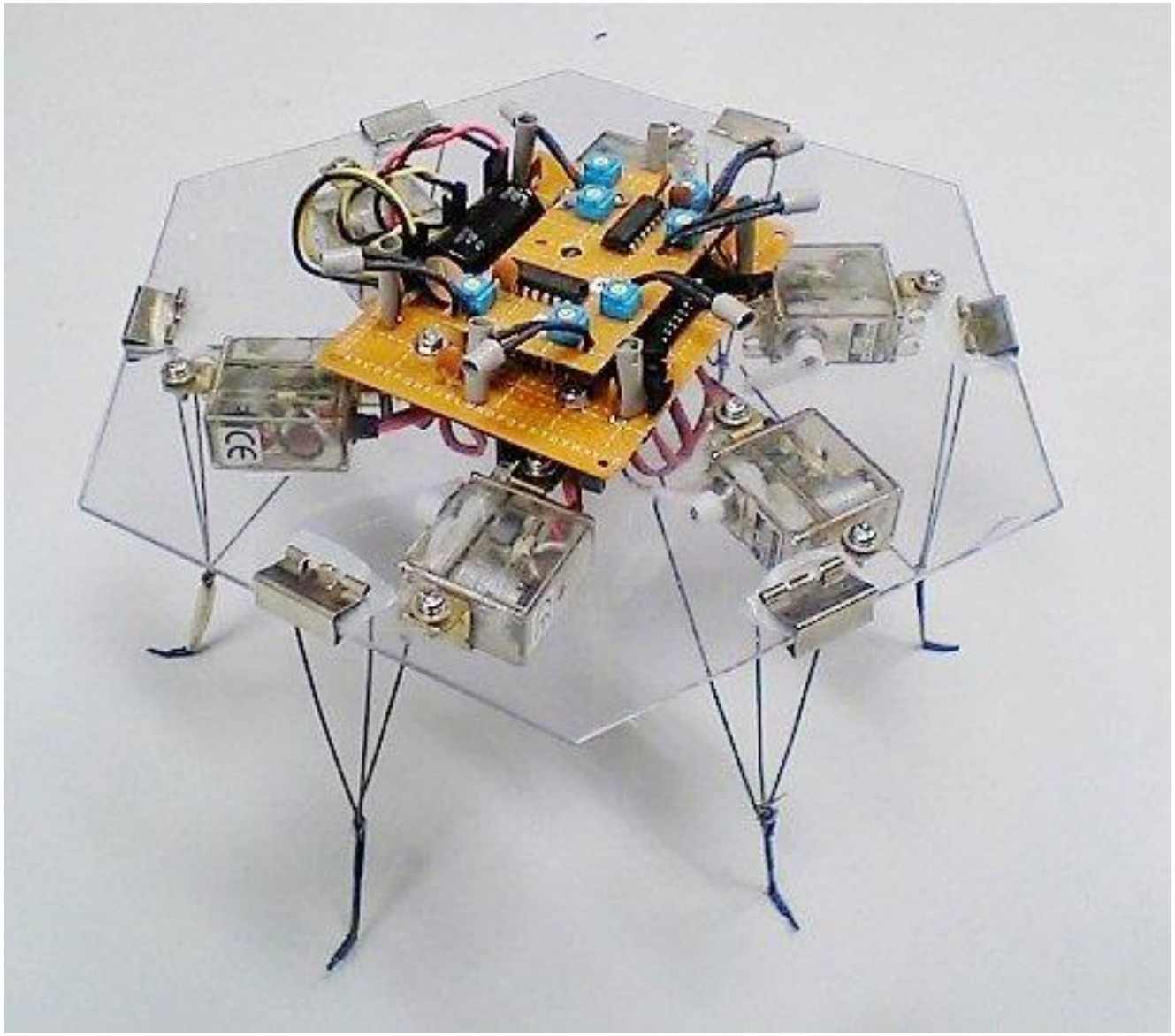
The pattern of failure has been especially striking in the computer industry. IBM dominated the mainframe market but missed by years the emergence of minicomputers, which were technologically much simpler than mainframes. Digital Equipment dominated the minicomputer market with innovations like its VAX architecture but missed the personal-computer market almost completely. Apple Computer led the world of personal computing and established the standard for user-friendly computing but lagged five years behind the leaders in bringing its portable computer to market.

Why is it that companies like these invest aggressively–and successfully–in the technologies necessary to retain their current customers but then fail to make certain other technological investments that customers of the future will demand? Undoubtedly, bureaucracy, arrogance, tired executive blood, poor planning, and short-term investment horizons have all played a role. But a more fundamental reason lies at the heart of the paradox: leading companies succumb to one of the most popular, and valuable, management dogmas. They stay close to their customers.

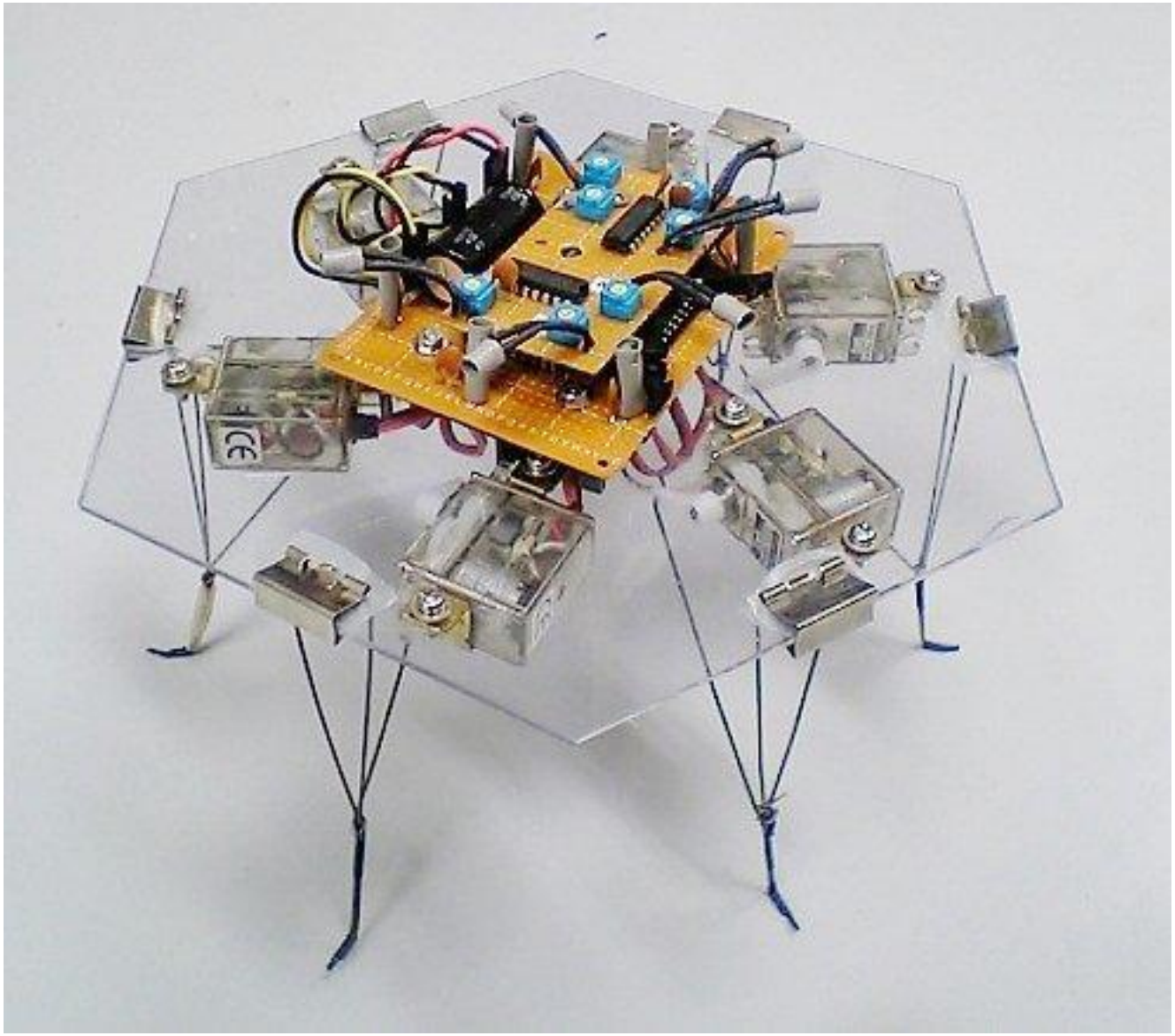
Although most managers like to think they are in control, customers wield extraordinary power in directing a company's investments. Before managers decide to launch a technology, develop a product, build a plant, or establish new channels of distribution, they must look to their customers first: Do their customers want it? How big will the market be? Will the investment be profitable? The more astutely managers ask and answer these questions,

Joseph L. Bower is the Donald Kirk David Professor of Business Administration at the Harvard Business School in Boston, Massachusetts. Clayton M. Christensen, an assistant professor at the Harvard Business School, specializes in managing the commercialization of advanced technology.

DRAWING BY CHRISTOPHER BING



K-P Zauner et al @ Southampton/Kobe, 2006



K-P Zauner et al @ Southampton/Kobe, 2006

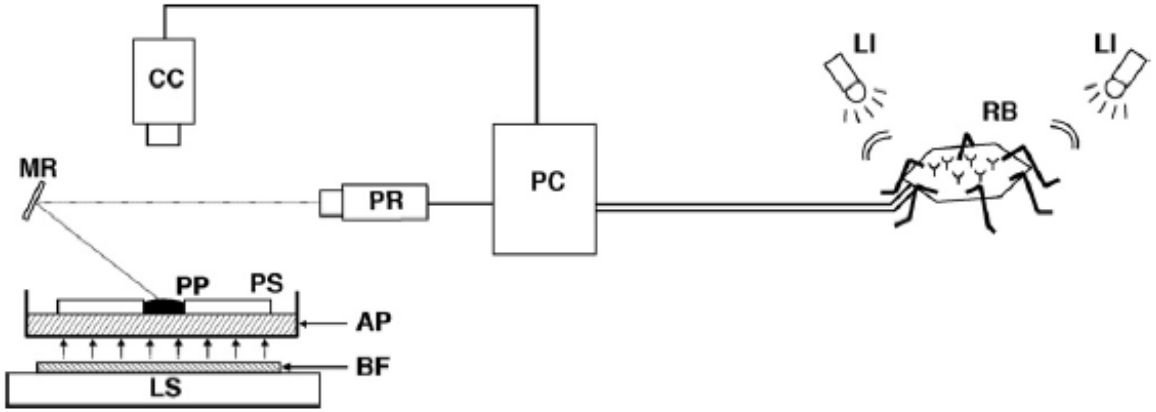


Fig. 5. Cellular robot controller. Sensors on the robot (RB) detect lamp input (LI) and transmit a signal to the computer (PC). The signals from the sensors are recoded by the PC into a spatial light pattern which is projected as light stimulus with a video projector (PR) via mirror (MR) onto the surface of the plasmodium (PP). Oscillations of the plasmodium (PP), which is patterned on an agar plate (AP) by a plastic sheet (PS), are detected by a CCD camera (CC) as intensity changes in light transmitted from the bandpass filtered (BF) light source (LS).

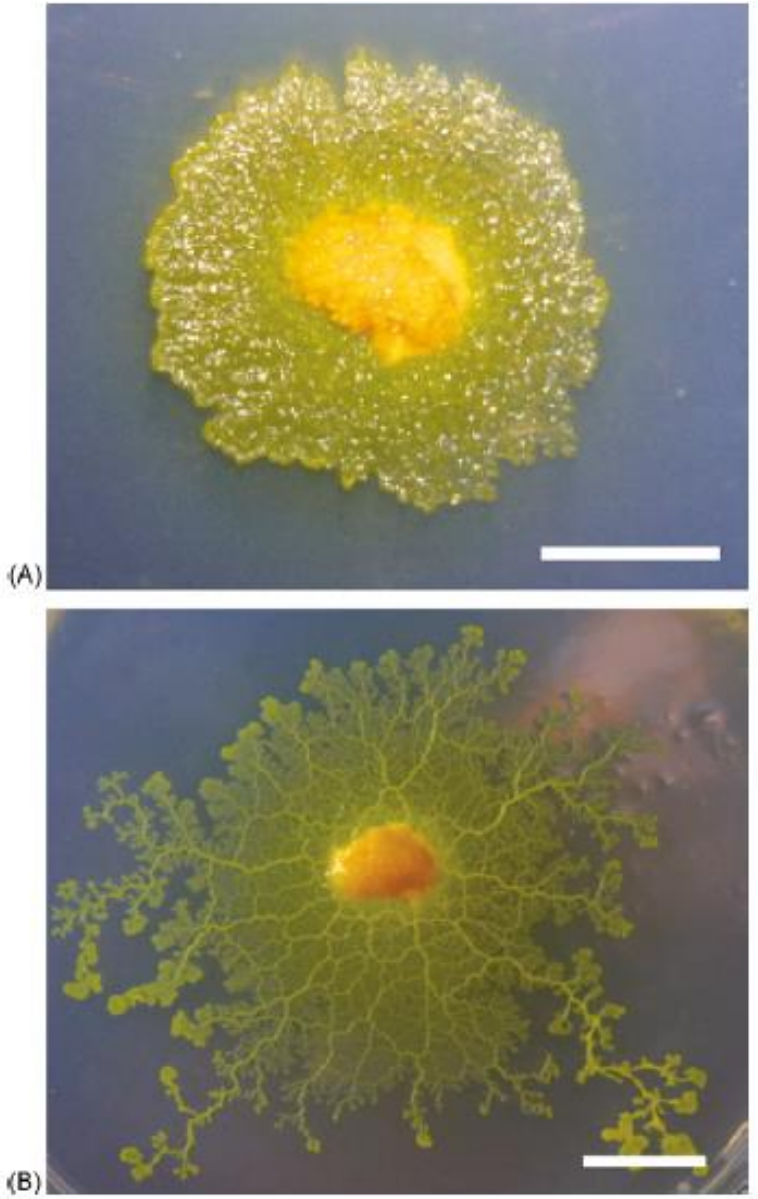


Fig. 1. Plasmodium of *Physarum polycephalum*. The early stage (A) and in developed form, approximately 8 h later (B). In both panels the length of the white bar corresponds to 10 mm.

K-P Zauner et al @ Southampton/Kobe, 2006

Available online at www.sciencedirect.com
ScienceDirect
BioSystems 87 (2007) 215–223
BioSystems
www.elsevier.com/locate/biosystems

Robot control with biological cells

Soichiro Tsuda[a,*], Klaus-Peter Zauner[b], Yukio-Pegio Gunji[a]

[a] *Graduate School of Science and Technology, Department of Earth & Planetary Science, Kobe University, Nada, Kobe 657-8501, Japan*
[b] *School of Electronics and Computer Science, University of Southampton, SO17 1BJ, United Kingdom*

Received 28 February 2005; received in revised form 8 July 2006; accepted 15 July 2006

Abstract

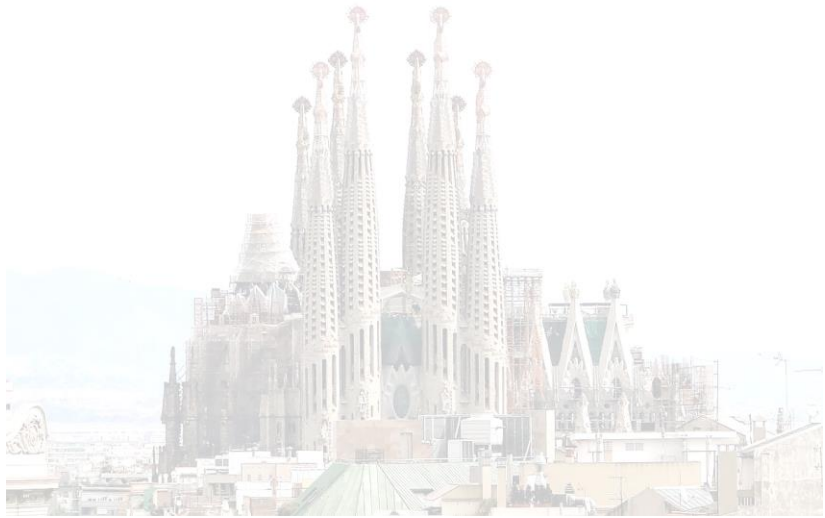
At present there exists a large gap in size, performance, adaptability and robustness between natural and artificial information processors for performing coherent perception-action tasks under real-time constraints. Even the simplest organisms have an enviable capability of coping with an unknown dynamic environment. Robots, in contrast, are still clumsy if confronted with such complexity. This paper presents a bio-hybrid architecture developed for exploring an alternate approach to the control of autonomous robots. Circuits prepared from amoeboid plasmodia of the slime mold *Physarum polycephalum* are interfaced with an omnidirectional hexapod robot. Sensory signals from the macro-physical environment of the robot are transduced to cellular scale and processed using the unique micro-physical features of intracellular information processing. Conversely, the response form the cellular computation is amplified to yield a macroscopic output action in the environment mediated through the robot's actuators.
© 2006 Elsevier Ireland Ltd. All rights reserved.

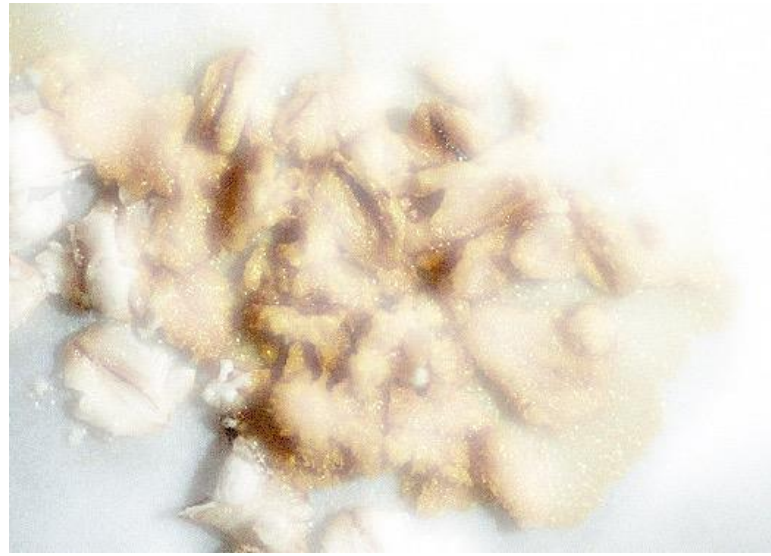
Keywords: Autonomous robots; Molecular computing; Coupled oscillators; Biologically inspired robotics

1. Introduction

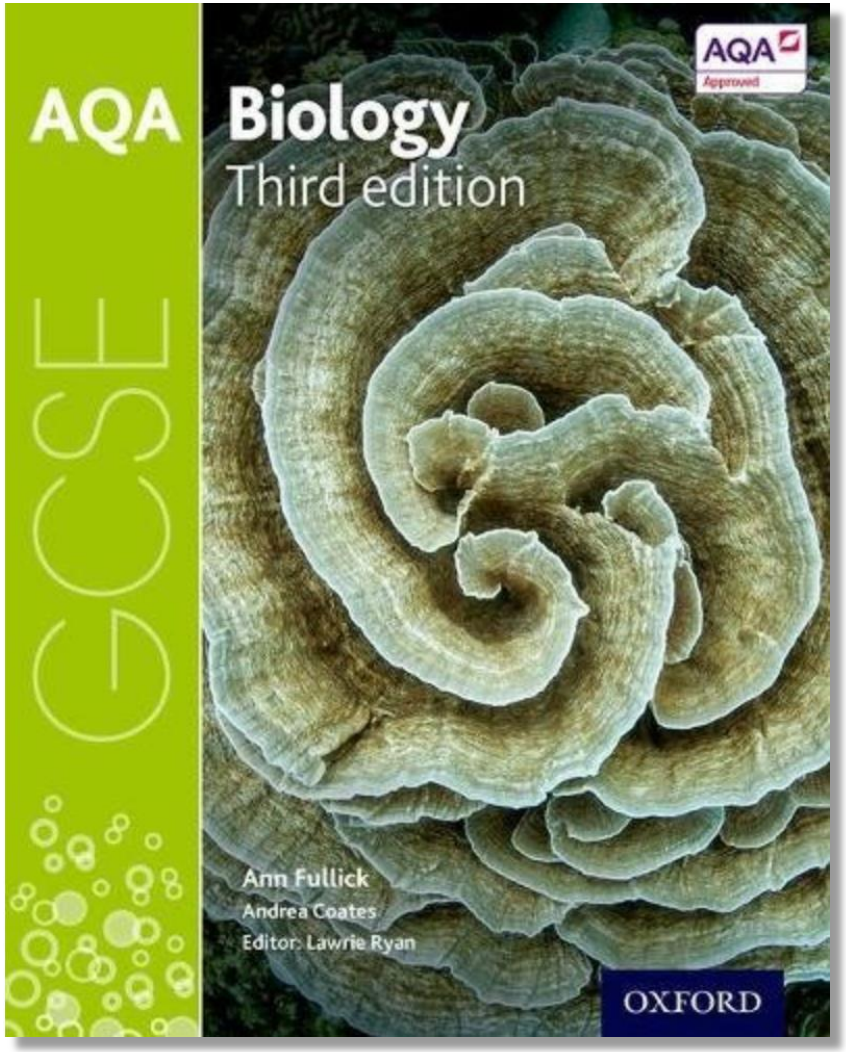
The prevalent approach to robot control is based on 'behaviour decomposition'. The interaction loop between robot and environment is decomposed and treated as individual modules. This concept simplifies controller

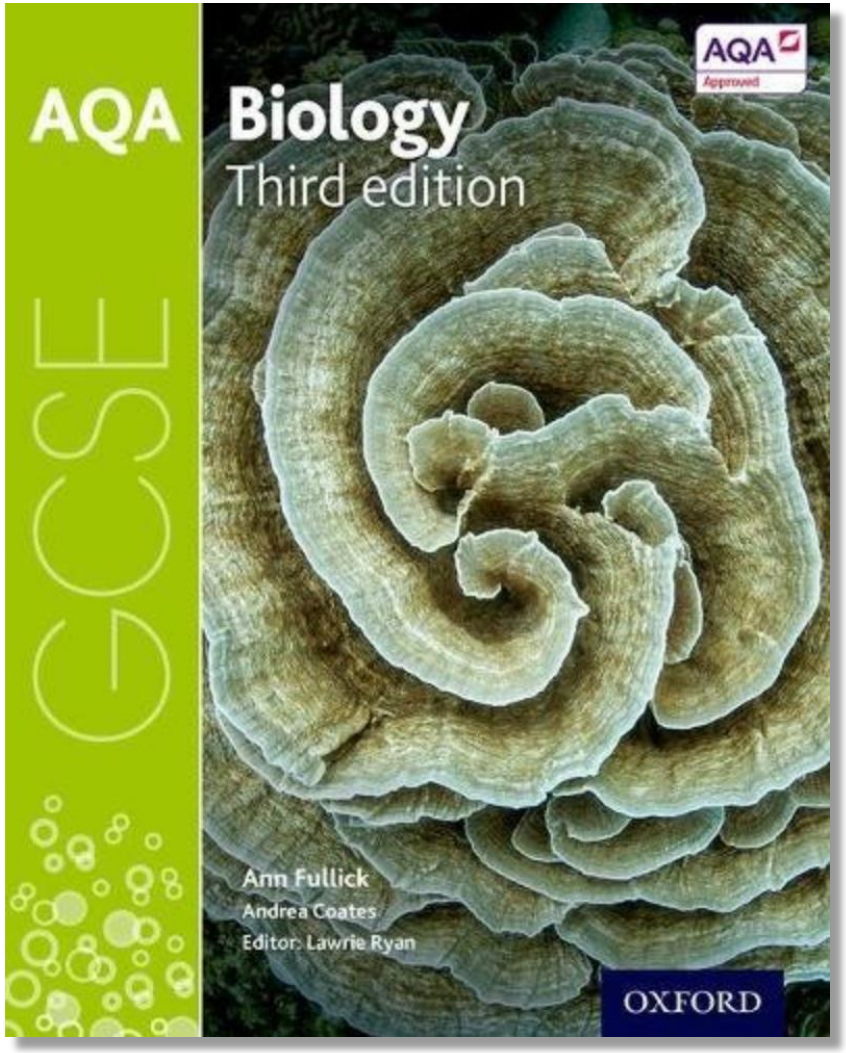
The use of 'behaviour decomposition' enables robots to successfully work in either a stable work space or with the support of a teacher. The scheme can be implemented in localised (Tani and Nolfi, 1999) or distributed fashion (Tani et al., 2004). Without a teacher,

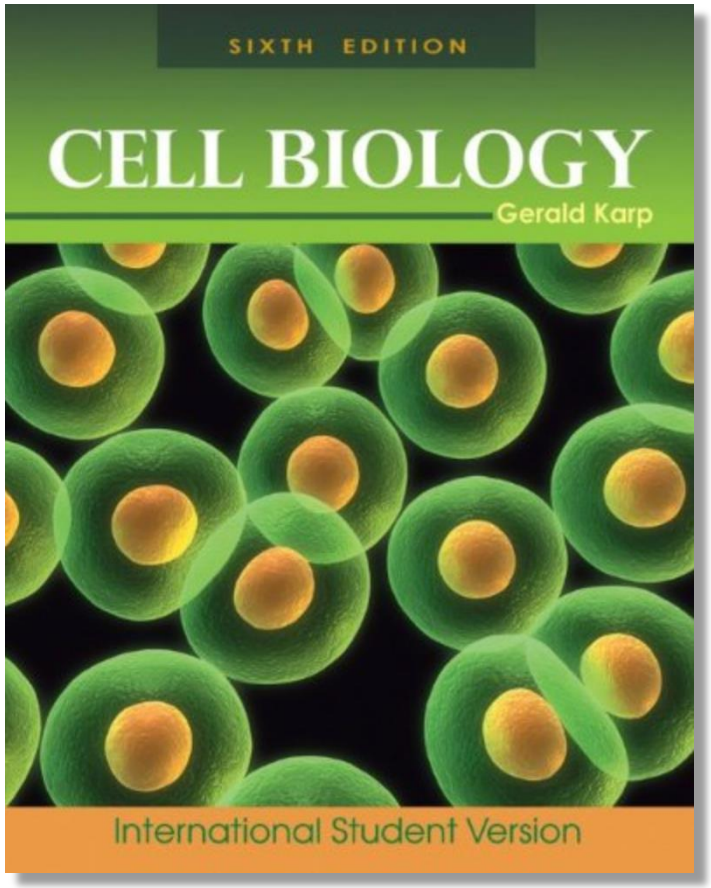


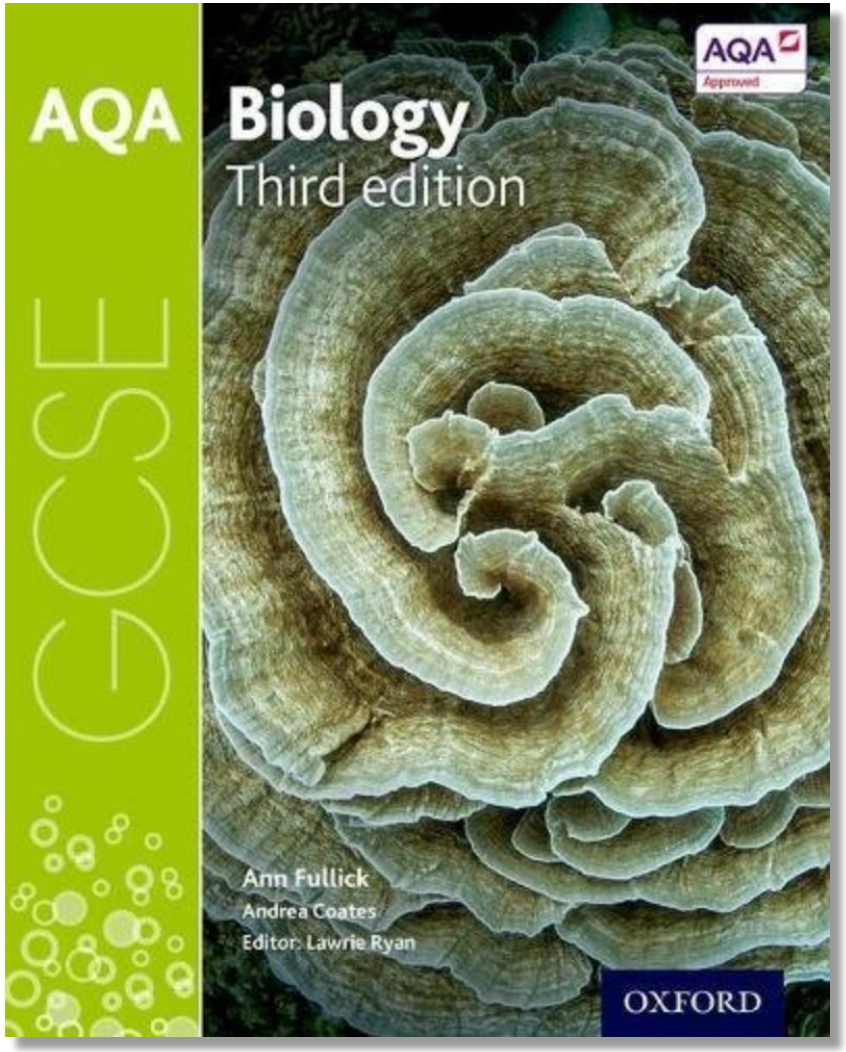


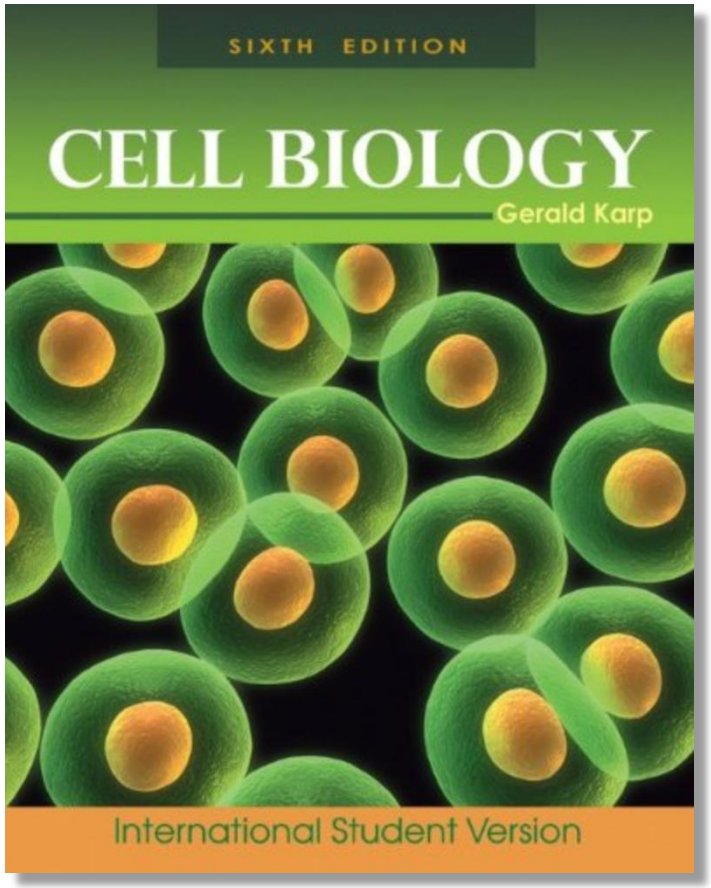
and FINALLY

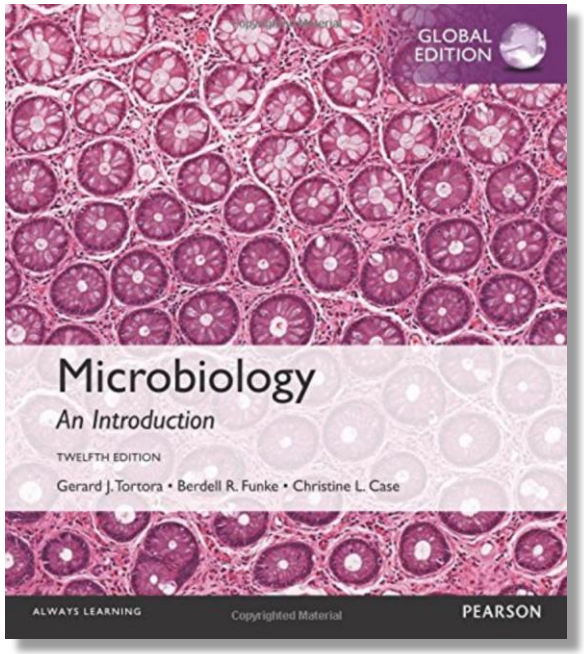

AQA
GCSE
AQA
Approved
Biology
Third edition
Ann Fullick
Andrea Coates
Editor: Lawrie Ryan
OXFORD

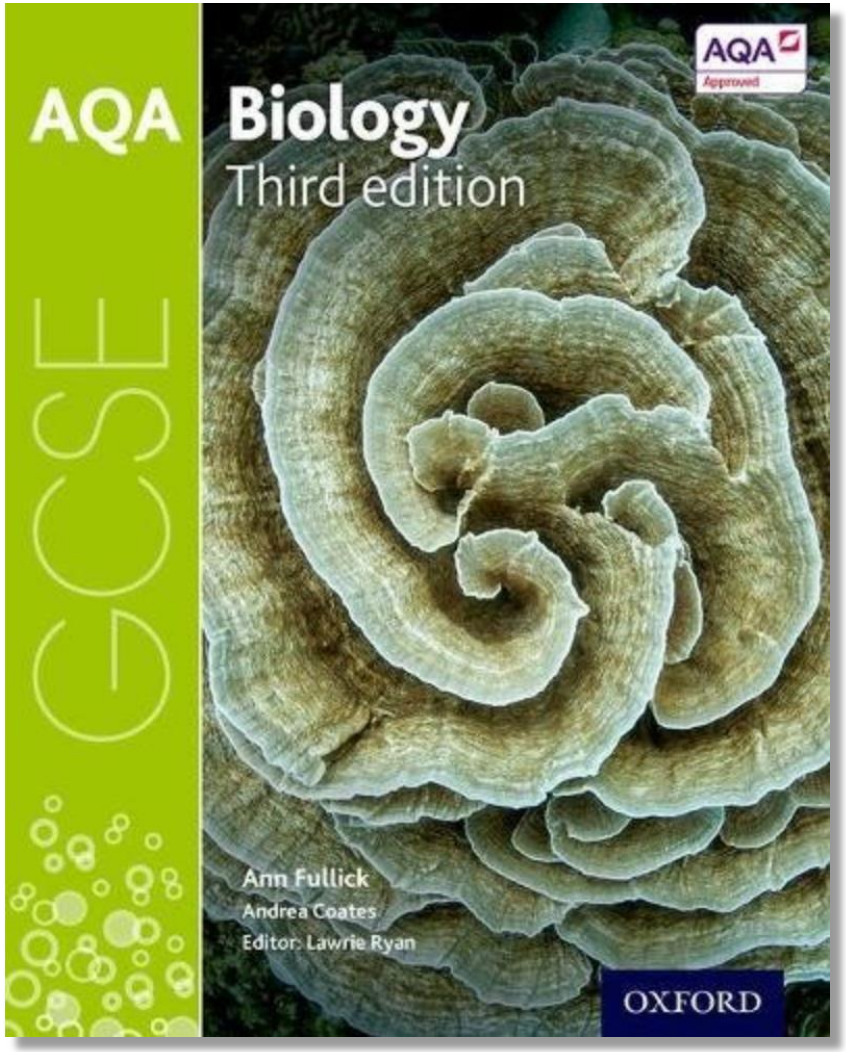

AQA
GCSE
Biology
Third edition
AQA
Approved
Ann Fullick
Andrea Coates
Editor: Lawrie Ryan
OXFORD

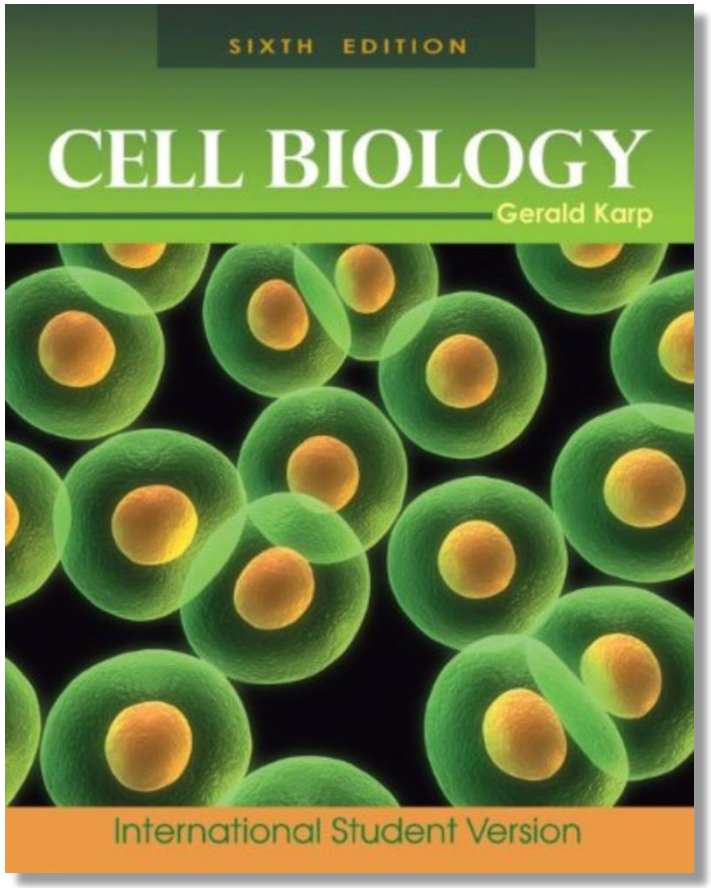

SIXTH EDITION
CELL BIOLOGY
Gerald Karp
International Student Version

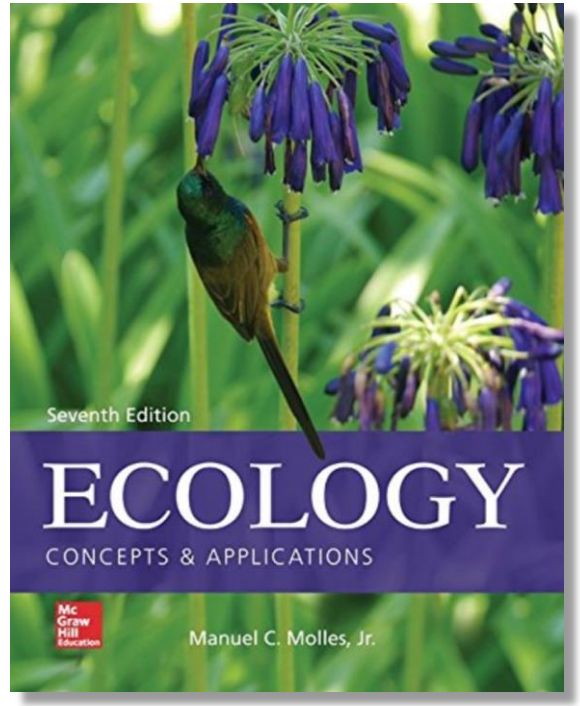

AQA
GCSE
Biology
Third edition
AQA
Approved
Ann Fullick
Andrea Coates
Editor: Lawrie Ryan
OXFORD

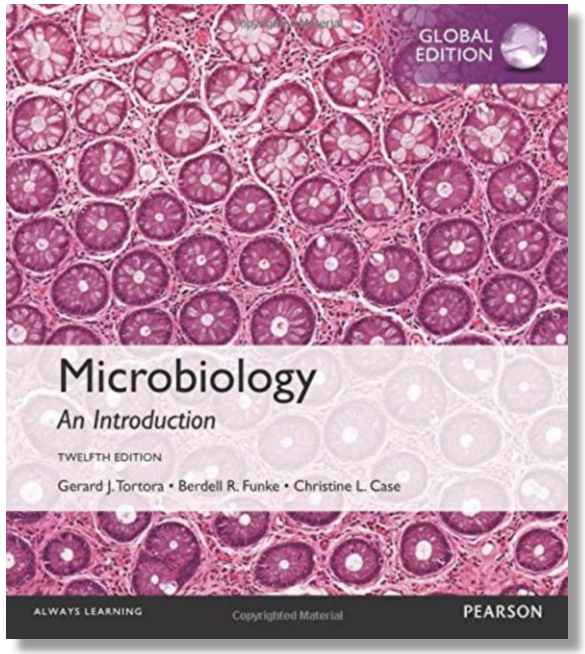

SIXTH EDITION
CELL BIOLOGY
Gerald Karp
International Student Version


GLOBAL EDITION
Microbiology
An Introduction
TWELFTH EDITION
Gerard J. Tortora • Berdell R. Funke • Christine L. Case
ALWAYS LEARNING
Copyrighted Material
PEARSON


AQA
GCSE
Biology
Third edition
AQA
Approved
Ann Fullick
Andrea Coates
Editor: Lawrie Ryan
OXFORD


SIXTH EDITION
CELL BIOLOGY
Gerald Karp
International Student Version


Seventh Edition
ECOLOGY
CONCEPTS & APPLICATIONS
Mc Graw Hill Education
Manuel C. Molles, Jr.


GLOBAL EDITION
Microbiology
An Introduction
TWELFTH EDITION
Gerard J. Tortora • Berdell R. Funke • Christine L. Case
ALWAYS LEARNING
Copyrighted Material
PEARSON

Thank You

Thank UoL

Le Fin

csdtc@bristol.ac.uk