the use of a pro-
the goal being a
based on devel-
d-in-hand. The
the teaching of

nount of detail,
ly inherent com-
he judicious use
d an underlying
ıg mathematics
some scientists,
erence rules are
is far from per-
thms have been
to develop new

tion of conven-
rover has been
ınguage Gypsy
ımmunications
uctive mathe-
:al proposition
programming.
ırl can extract
truction.

Computer Systems

Forest Baskett David Clark
A. Nico Habermann Barbara Liskov
Fred B. Schneider Burton Smith

Research in computing systems is concerned with developing principles and tools to enhance the accessibility and power of computers. The goal is to bridge the gap between the demand for computing imposed by applications and the supply provided by extant technologies. Research is performed by designing systems, analyzing systems, and—most importantly—abstracting key problems and devising mechanisms to facilitate their solution. Thus, in addition to producing systems, researchers make important contributions by inventing new abstractions and devising ways to manipulate them.

The nature of computing systems research is diverse. As in more traditional sciences, there is an empirical element. Programs do not behave in random ways, and knowledge of how they do behave allows the structures that execute them to be optimized. Unlike the natural sciences, however, the phenomena being studied are of our own making. Changing the phenomena—that is, the hardware or software—is a perfectly acceptable way to avoid a problem. Finally, computing systems research has a significant engineering component. Computing systems tend to be complicated enough that building them is the only way to evaluate certain ideas.

We cannot hope to survey research in computing systems in a short space. The field is enormous, spanning aspects of hardware and software design. However, it is possible to get a taste of computing systems research by looking at a few of its scientific contributions. We do this as follows. The examples selected are representative of the type and style of research in the area, although they reflect our biases.

Operating Systems

Sharing is one way to make an expensive resource, like a computing system, more accessible to a collection of users. To put this principle into practice,

called *operating*
›s to be shared
ıaving a virtual
.s not a simple
revent different
use of available

user programs.
h received con-
, designing in-
what state an
:onsequently it
correct thing.
!
form of asyn-
ınd an elegant
nd algorithms
ıes. These, in
ıt of others to
›perties of so-
ıd with a par-
›e formulated
.though mem-
ion, computer
ıic. Some of
ing operating
The existence
ıble as (then)
chnology and
that does not

.d impact be-
an active re-
and compu-
ımantics and
lism. In ad-
›dal logics is
ıe suitability

Sharing a computer among users involves more than just sharing processor cycles among programs. To maintain the illusion that each user has a private machine, memory usage must also be controlled. This is because programs written in isolation might use identical names to designate (what should be) different memory locations. One solution to this problem, developed by computing systems researchers, was *virtual memory*, an abstraction that, like real memory, maps names to memory locations. Virtual memory is managed by the operating system. Tables are maintained describing the mapping in effect, and information (in units called pages) is moved from relatively slow peripheral storage devices to main memory and back as it is needed by executing programs. As a result, it is possible for each user's virtual machine to behave as if it has a private memory that is larger than the one actually connected to the processor.

Virtual memory makes it possible to write programs without concern for the actual memory size of a computer. It illustrates a pervasive theme in computing systems research: defining and implementing abstractions that package technology for easier use. As another example, file and data-base systems implement abstractions that allow information to be saved and retrieved without concern for how or where the information is stored. In fact, the success of many operating systems, including the popular UNIX system, can be attributed directly to the clean, high-level abstractions they present to their users. These abstractions allow programmers to devote more time to their applications and less time to dealing with idiosyncrasies of the system. Inventing clean, high-level abstractions is difficult, but then, designing tools that will be used for as yet unimagined tasks is always hard.

Virtual memory illustrates another important aspect of computer systems research: controlling resource use to achieve high utilization. To load a new page into main memory, we might have to move back an old page to the peripheral storage device. The obvious question is which page to evict. Empirical studies of program behavior reveal that page references are not entirely random. Programs tend to exhibit locality: a recently referenced page is more likely to be referenced sooner than a page that has not been referenced for some time. Locality can be exploited in designing a page replacement algorithm, allowing a virtual memory system to perform considerably better than if page-out decisions were made randomly. Here, experimental evidence concerning program behavior allowed an aspect of the system to be optimized.

Processor cycles and memory pages are just two examples from a diverse collection of resources that an operating system must manage. To ensure adequate system performance, access to all of these resources must be controlled. In early systems, scheduling policies drew heavily from previous work in telephony. However, it soon became apparent that new policies were required. Resource requests in computing systems (unlike telephone systems) are characterized by high-variance distributions, as exemplified by the rule that a large percentage of a resource will be required by a small number of the requesters. To ensure high resource utilization with these high-variance distributions, researchers derived new scheduling and allocation methods. Contributions to traditional job-shop scheduling theory were also made. Problems in analyzing computing systems led to advances in traditional operations research methods, including new methods for solving queuing models.

One might think that today's personal computers render much of this work obsolete, since machines are no longer shared by users. Just the contrary is true. First, expensive resources, like high-resolution printers and file storage systems, must still be shared. Second, most personal computers allow a user to run several programs concurrently. Window facilities, for example, allow the user to interact concurrently with a number of programs, thereby supporting a synergism between these tools. Although the processor is not being shared among users, it is being shared among these tasks, and the theory and solutions developed for implementing multiuser time-sharing systems still apply.

Computer Architecture

The needs of some applications are defined in terms of cost and raw computing, communication, and storage capacity. Computer architecture is the area of computing systems research concerned with designing computers to satisfy those needs. The term *architecture* is particularly appropriate because design is done at a fairly high level of abstraction. At any given time, current technology provides building blocks—arithmetic, logic, and storage elements—with certain performance characteristics. The computer architect attempts to combine these building blocks so that they cooperate harmoniously to implement a structure that satisfies performance and cost goals. By intelligent choice of abstractions and/or their implementation, we can exploit the latest technological breakthrough or overcome a technological bottleneck.

One example of an innovation in the implementation of an abstraction is the idea of instruction pipelining. The machine language programmer

is told—that
occurs as a se

1. The inst
2. The inst
3. Operan
4. The ope
5. Results

The naive im
one instructio
sons, accessin
units of mem
struction wou
of steps 1, 3,
2 and 4. A
plete instruct
assembly line
between adja
decoded, the
can be compl
busy all the t

The idea
be applied at
arithmetic un
accomplished
level of the s
achieve extre
lection of fun
the data tran

Caching is
high perform
nological one
has limited ba
memory. The
performance

s must be con-
from previous
ıt new policies
ılike telephone
exemplified by
:ed by a small
ion with these
ng and alloca-
ıg theory were
o advances in
›ds for solving

much of this
Just the con-
ı printers and
ıal computers
facilities, for
: of programs,
the processor
›se tasks, and
time-sharing

nd raw com-
:ecture is the
:omputers to
›ropriate be-
y given time,
and storage
.ter architect
:rate harmo-
d cost goals.
:ion, we can
echnological

abstraction
›rogrammer

is told—that is, provided with the abstraction—that instruction execution occurs as a sequence of five operations:

1. The instruction is fetched from memory.
2. The instruction is decoded.
3. Operands are fetched from memory.
4. The operands are combined as prescribed by the instruction.
5. Results are stored in memory.

The naive implementation of this five-operation cycle would be to process one instruction completely, then the next, and so on. For technological reasons, accessing memory usually involves a significant delay. Measured in units of memory-access delay, the elapsed time between completing each instruction would be at least 3 memory access times—one access time for each of steps 1, 3, and 5. Notice, also, that the memory is left idle during steps 2 and 4. A *pipelined* implementation of instruction execution could complete instructions at close to twice this rate. A pipeline works just like an assembly line and exploits the empirically observed absence of dependencies between adjacent instructions. In this case, while one instruction is being decoded, the next is fetched, and so on. The result is that an instruction can be completed every 3 memory-access delays. Moreover, memory is kept busy all the time.

The idea of pipelining is not restricted to instruction processing. It can be applied at various levels in the design of a computing system. Pipelined arithmetic units enable time-consuming operations, like multiplication, to be accomplished at a high rate. Pipelining can also be applied at the highest level of the system. A class of computers, called *systolic arrays*, is able to achieve extremely high throughput because these computers consist of a collection of functional units connected into one large pipeline that implements the data transformation described by a program.

Caching is another example of how computer architects are able to achieve high performance by conceptual breakthroughs rather than relying on technological ones. High-speed memory tends to be expensive, and all memory has limited bandwidth. Ideally, we desire cheap, high-speed, high-bandwidth memory. The question is how to implement a memory abstraction with these performance characteristics; caching provides the answer. A *cache* is a small, high-speed memory that serves as a front end to a larger, slow-speed memory. Information is moved into the cache when it is needed and moved out when it is no longer needed. Because programs exhibit locality in memory references, the cost of moving information into a cache can be amortized, and most memory accesses will be to the high-speed cache. Caches that are smaller than 1 percent of the total memory size are sufficient for dramatic performance improvements (i.e., over 95 percent memory accesses can be satisfied by the cache). Thus, a relatively high-speed memory is implemented by using a large amount of slow-speed memory, a small amount of high-speed memory, and mechanisms to transfer information between them.

The issues in implementing a cache are similar to those associated with implementing a virtual memory. It is not rare for both hardware and software to involve the same abstractions. Consequently, both can benefit from new abstractions and innovations in implementing old ones. In fact, the computer architect often must choose between realizing an abstraction directly in hardware and choosing more primitive abstractions to implement, leaving it to systems software to realize that abstraction. This also means, however, that work in computer architecture can be responsible for new research directions in programming languages, compiling, and algorithms. For example, recent investigations into reduced instruction set computers (RISC), which are based on the premise that small and simple instruction sets are best, have stimulated research into compiler techniques.

Most agree that still more computing power will be needed to execute applications programs that unravel new complexities in, for example, particle physics, quantum chemistry, and genetics and to handle engineering applications such as computer-aided design and computer-controlled manufacturing. Future computers will meet these goals not only by innovation in electronic component technology but also by an architectural revolution—altering the instruction processing abstraction from the single-stream (one instruction at a time) Von Neumann model to a model in which multiple instruction streams are processed in parallel. The correct choice of parallel processing abstraction, if indeed there is a correct choice, remains a hotly debated topic. Past work from operating systems gives some insight into parallel processing, but only for instruction execution abstractions that correspond to parallel asynchronous processes. Other computational models have also been proposed. Examples of such abstractions are the data-flow and functional programming models. In both, a set of equations of a particular form, rather than a sequence of instructions, defines a computation.

As a result, the in
created as the com
Most of the pro
allel processes are
architectures emp
plementing this a
ory because the n
schemes, in which
might reside conc
tigated as a way t
abstraction under
abstraction requi
For the extremel
logically infeasib
the topology of
programs in whi
topology and the
Additional re
tions will be pro
The experiment
computing syste
does not suffice
significantly fro
constructed and
computers are u
could be simula
ulation is just t
can be great. A

Networks ar

Not all applica
cost. Sometim
system, usuall
example, an o
on computers
these location
in which place
cessors must

slow-speed mem-
l and moved out
:ality in memory
ın be amortized,
ıe. Caches that
ıfficient for dra-
ory accesses can
ıemory is imple-
small amount of
ı between them.
associated with
dware and soft-
an benefit from
s. In fact, the
abstraction di-
; to implement,
his also means,
nsible for new
nd algorithms.
set computers
ple instruction
ıes.
led to execute
example, par-
le engineering
ıtrolled manu-
innovation in
l revolution—
e-stream (one
hich multiple
ice of parallel
nains a hotly
e insight into
ions that cor-
ional models
the data-flow
ons of a par-
omputation.

As a result, the instruction stream is not defined prior to execution but is created as the computation proceeds.

Most of the problems associated with parallel architectures based on parallel processes are associated with the communications abstraction. Some architectures employ a shared memory for communication. However, implementing this abstraction requires an extremely high bandwidth memory because the memory must service many (fast) processors. New caching schemes, in which each processor has its own cache and program variables might reside concurrently in more than one cache, are currently being investigated as a way to solve this bandwidth problem. A second communication abstraction under investigation is the use of message passing. Realizing this abstraction requires communications channels that interconnect processors. For the extremely large numbers of processors contemplated, it is technologically infeasible to connect every pair with a direct link. Yet, restricting the topology of interconnections makes an architecture poorly suited for programs in which there is a clash between the processor interconnection topology and the program's communications topology.

Additional research will be required to resolve these problems. Abstractions will be proposed, and experiments will affirm or deny their viability. The experiments will be costly because designing and engineering a large computing system can require significant time and resources. Moreover, it does not suffice to build just the hardware. For abstractions that differ significantly from those with which we have experience, software must be constructed and programming methods must be developed. And, although computers are universal machines and therefore any new computing system could be simulated, it is still necessary to build the hardware because simulation is just too slow to gain meaningful experience. However, the payoff can be great. All the sciences can benefit from the tools that result.

Networks and Distributed Systems

Not all application requirements can be translated into access, capacity; and cost. Sometimes, an application imposes constraints on the structure of a system, usually in terms of physical placement of system components. For example, an organization might need to share information that is located on computers at various branch locations, requiring that the computers at these locations be linked. Process control applications are another example in which placement of computers is dictated by the application. Here, processors must be located near the sensors and actuators that interact with the physical process being controlled. Finally, implementing the fault tolerance needed for high availability requires that when components fail, they do so independently. Processors that are physically separated and linked by a communications network exhibit this failure independence.

A distributed system is a collection of computers interconnected using a computer communications network that provides (relatively) narrow-bandwidth, high-latency communications channels. At first, we might think that an understanding of telephony would be sufficient to enable networks to be implemented and that an understanding of parallel asynchronous processes (from operating systems) would be sufficient to enable applications to be designed for distributed systems. Unfortunately, computer communications demands are not well served by traditional telephone switching technology. In addition, the use of narrow-bandwidth, high-latency communications channels changes the coordination problems that can and must be solved, in addition to changing the costs of various solutions. Computing systems research in networks and distributed systems has tackled these problems, in some cases uncovering fundamental limitations and results in fault tolerance and coordination of communicating processes.

The central proposition of modern data networking is that a useful communications abstraction can be implemented using *packet switching* (as opposed to circuit switching). In packet switching, a communications channel is multiplexed serially by sending small *packets.* Each packet contains routing information and a small element of data. Information to be transferred between two sites is decomposed into packets, each of which independently wends its way to the destination, perhaps even taking different routes. The use of packets permits a higher rate of multiplexing, and hence a high degree of sharing of expensive communications bandwidth. Even telephone companies have now embraced packet switching to implement the circuit-switching abstraction they present to their customers.

Packet switching does not itself prevent the offered load from exceeding available bandwidth; nor does it prevent statistical fluctuations in load from causing short-term overloads. These congestion problems have occupied network designers for the last century. Algorithms designed to control congestion are difficult to construct because congestion might arise and disperse faster than the system can respond. A good analogy here is with air-traffic control. Imagine the difficulty of managing the air-traffic control system if controllers could only communicate by putting messages into airplanes and flying them from airport to airport! Actual air-traffic controllers have access

to low-delay co
computer comr
In addition
relative to the
the system. As
in different or
have developec
phenomenon, r
cations for log
nature of caus
to find that tir
and that anom
distributed sys
Another m
for the commu
packet deliver
lose or reorder
are delivered i
this abstractic
lays. For this
been investiga
cause of the g
network. Cha
as six orders
One outg
increased unc
most centrali
running if a
tributed syst
even if other
function and
fault toleranc
The tradi
sures, like M
tolerance, ch
t of failures
Clearly, stoc
stochastic ch
of t fault tol

; the fault toler-
›nents fail, they
d and linked by
.
›rconnected us-
.tively) narrow-
we might think
nable networks
nchronous pro-
›le applications
ıputer commu-
ıone switching
ı-latency com-
; can and must
ions. Comput-
; tackled these
and results in

; a useful com-
itching (as op-
ations channel
contains rout-
be transferred
independently
ıt routes. The
a high degree
phone compa-
:uit-switching

om exceeding
; in load from
occupied net-
›ntrol conges-
and disperse
ith air-traffic
:rol system if
ırplanes and
s have access

to low-delay communications paths (e.g., telephone and radio) for control; a computer communications network must use its own data paths for control.

In addition, process execution speeds in a distributed system are high, relative to the speed with which processes can change or sense the state of the system. As a result, system processes can observe the same set of events in different orders, unless those events are causally related. Researchers have developed theories to reason about computing systems subject to this phenomenon, resulting in new abstractions about computation, new applications for logics of knowledge and belief, and even new insight into the nature of causality. Those familiar with special relativity are not surprised to find that time-space diagrams have received application in these theories and that anomalies predicted by special relativity are actually observable in distributed systems.

Another major concern to networking researchers is the correct choice for the communications abstraction provided to users of the network. Raw packet delivery is unsatisfactory as an abstraction because the network may lose or reorder packets. The *virtual circuit* abstraction ensures that packets are delivered in the order sent; unfortunately, it is impossible to implement this abstraction without admitting the possibility of unbounded delivery delays. For this reason, both less powerful and more powerful abstractions have been investigated. Devising suitable abstractions is particularly difficult because of the great variation that can be found in the channels composing a network. Channels can vary in bandwidth and end-to-end delay by as much as six orders of magnitude.

One outgrowth of research in networks and distributed systems is an increased understanding of fundamentals for achieving fault tolerance. In most centralized systems, no serious attempt was made to keep the system running if a component failed, and the whole system failed as one. In distributed systems, it became desirable to keep parts of the system running, even if other parts failed. This has resulted in new methods for partitioning function and responsibility and a new set of abstractions for thinking about fault tolerance.

The traditional view of fault tolerance is in terms of stochastic measures, like MTBF (mean time between failures). A more recent view, t-fault tolerance, characterizes fault tolerance in terms of the maximum number t of failures that can occur before the system will violate its specification. Clearly, stochastic measures can be derived from t fault tolerance if given stochastic characterizations for system component failures. The advantage of t fault tolerance is that it permits evaluation of fault tolerance that is achieved through design—independent of the reliability characteristics of the components used to implement that design. Fault tolerance can now be viewed in a technology-independent manner. This view of fault tolerance has led to some surprising insights. One is that distributing an input or coordinating the actions of the components in a replicated system can be very expensive and sometimes impossible. Results associated with the so-called Byzantine Generals problem give bounds on the costs. The results also give insight into types of failure that less expensive implementations are unable to tolerate. They show that TMR (triple-modular redundancy), a widely used fault tolerance technique in computer engineering, is based on some previously unstated assumptions that are not always valid.

Computer scientists have enjoyed the benefits of computer networking and distributed computation since the 1960s, when the ARPANet was first constructed. Initially developed as an experimental network connecting computer science research facilities, the network is now widely used by computer scientists (and others) for day-to-day communications activities, including electronic mail, remote login, and file transfer and sharing. Electronic mail has revolutionized communications patterns among users and has led to commercial ventures and standards of use in all disciplines. Office automation and exploitation of personal computers are possible only because of our understanding of computer communications networks and distributed systems. And, mundane as they are, these largely clerical uses, made possible by computing systems research, are changing the way business is conducted, financial resources are managed, and people conduct their daily lives.

Concluding Remarks

Computing systems research, like much of computer science, is about properties and implementation of abstractions. The ultimate utility of these abstractions derives from the extent to which they make computing resources available to applications. A reasonably small set of abstractions—concerned with cooperation and sharing resources—finds utility over and over at various levels of a computing system, leading one to believe that these abstractions are, in some sense, fundamental. New breakthroughs in electronic component technology can be exploited and new applications demands sated with minimal disruption to users' views of computing by virtue of these abstractions.

Although there is a large engineering component in computing systems research—one builds systems in order to understand the utility of the ab-

stractions they
primary researc
observe, predict
those explanati
research is the

ıaracteristics of
ınce can now be
' fault tolerance
an input or co-
;em can be very
th the so-called
results also give
ions are unable
ıncy), a widely
based on some

ıter networking
ˀANet was first
onnecting com-
ɜd by computer
ities, including
Electronic mail
ınd has led to
Office automa-
because of our
istributed sys-
made possible
s is conducted,
.ily lives.

is about prop-
ty of these ab-
:ting resources
ns—concerned
ɔver at various
e abstractions
tronic compo-
ıds sated with
these abstrac-

uting systems
ity of the ab-

stractions they implement—it would be a mistake to view engineering as the primary research activity. The main contribution of the natural sciences is to observe, predict, and explain phenomena, not the experiments that validate those explanations. Similarly, the main contribution of computing systems research is the abstractions and our understanding of why they work.