COMMUNICATIONS OF THE ACM

CACM.ACM.ORG

05/2009 VOL.52 NO.05

Security in the Browser

Spending Moore's Dividend

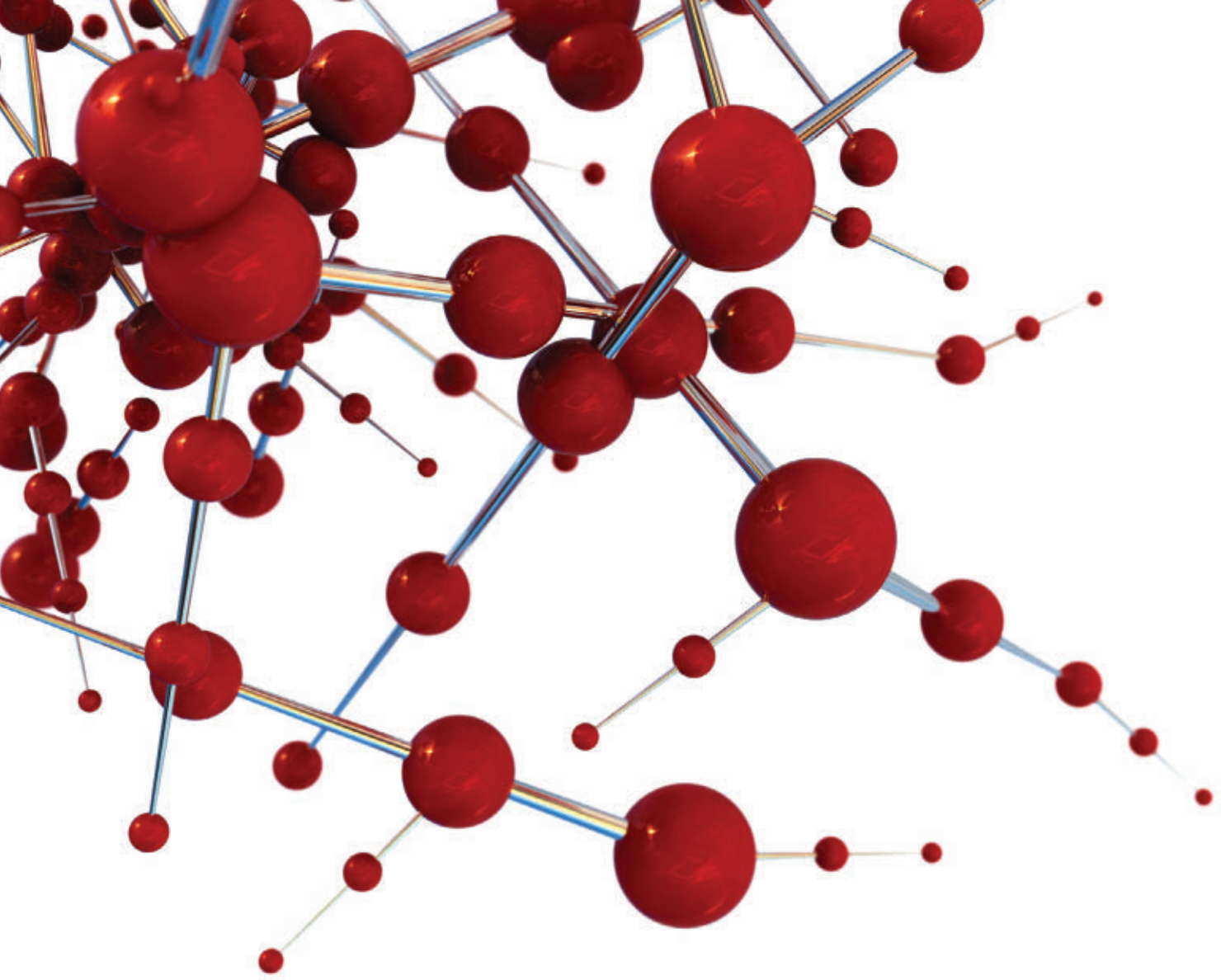
Algorithmic Systems Biology

Computing Needs Time

Rethinking Signal Processing

The Net Neutrality Debate Hits Europe

Association for Computing Machinery



THE ACM A. M. TURING AWARD

ACM, INTEL, AND GOOGLE CONGRATULATE **BARBARA H. LISKOV** FOR HER FOUNDATIONAL INNOVATIONS IN PROGRAMMING LANGUAGE DESIGN THAT HAVE MADE SOFTWARE MORE RELIABLE AND HER MANY CONTRIBUTIONS TO BUILDING AND INFLUENCING THE PERVASIVE COMPUTER SYSTEMS THAT POWER DAILY LIFE.

BY THE COMMUNITY...

FROM THE COMMUNITY...

FOR THE COMMUNITY...

"Intel is a proud sponsor of the ACM A. M. Turing Award, and is pleased to join the community in congratulating this year's recipient, Professor Barbara Liskov. Her contributions lie at the foundation of all modern programming languages and complex distributed software. Barbara's work consistently reflects rigorous problem formulation and sound mathematics, a potent combination she used to create lasting solutions."

Andrew A. Chien
Vice President, Corporate Technology Group
Director, Intel Research

For more information see www.intel.com/research.

"Google is delighted to help recognize Professor Liskov for her research contributions in the areas of data abstraction, modular architectures, and distributed computing fundamentals—areas of fundamental importance to Google. We are proud to be a sponsor of the ACM A. M. Turing Award to recognize and encourage the research that is essential not only to computer science, but to all the fields that depend on its continued advancement."

Alfred Z. Spector
Vice President, Research and Special Initiatives, Google

For more information, see http://www.google.com/corporate/index.html and http://research.google.com/.

Financial support for the ACM A. M. Turing Award is provided by Intel Corporation and Google.

COMMUNICATIONS OF THE ACM

Departments

Last Byte

News

Education is fast becoming a global affair.

Viewpoints

PHOTOGRAPH BY ANDY McNEILL

Association for Computing Machinery
Advancing Computing as a Science & Profession

05/2009
VOL. 52 NO. 05

Practice

Article development led by acmqueue
queue.acm.org

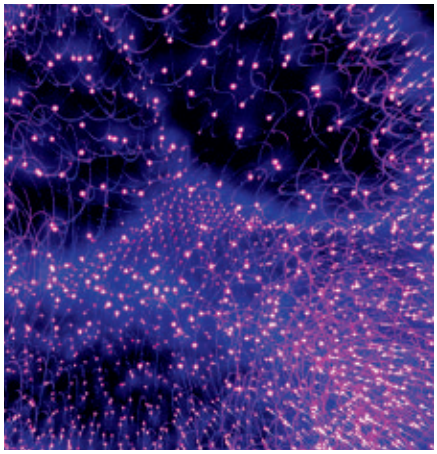
Contributed Articles

About the Cover:
Users want a browser to be as safe as a vault, but they also want usability features that compromise its security. Can we find a happy—and effective—balance?

Illustration by Jonathan Barkat.

ILLUSTRATION BY ROBERT HODGIN

Review Articles



Research Highlights

Virtual Extension

As with all magazines, page limitations often prevent the publication of articles that might otherwise be included in the print edition. To ensure timely publication, ACM created *Communications*' Virtual Extension (VE).

VE articles undergo the same rigorous review process as those in the print edition and are accepted for publication on their merit. These articles are now available to ACM members in the Digital Library.

COMMUNICATIONS OF THE ACM

A monthly publication of ACM Media

Communications of the ACM is the leading monthly print and online magazine for the computing and information technology fields. *Communications* is recognized as the most trusted and knowledgeable source of industry information for today's computing professional. *Communications* brings its readership in-depth coverage of emerging areas of computer science, new trends in information technology, and practical applications. Industry leaders use *Communications* as a platform to present and debate various technology implications, public policies, engineering challenges, and market trends. The prestige and unmatched reputation that *Communications of the ACM* enjoys today is built upon a 50-year commitment to high-quality editorial content and a steadfast dedication to advancing the arts, sciences, and applications of information technology.

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and profession. ACM provides the computing field's premier Digital Library and serves its members and the computing profession with leading-edge publications, conferences, and career resources.

BPA Audit Pending

Subscriptions
Annual subscription cost is included in the society member dues of $99.00 (for students, cost is included in $42.00 dues); the nonmember annual subscription rate is $100.00.

ACM Media Advertising Policy
Communications of the ACM and other ACM Media publications accept advertising in both print and electronic formats. All advertising in ACM Media publications is at the discretion of ACM and is intended to provide financial support for the various activities and services for ACM members. Current Advertising Rates can be found by visiting http://www.acm-media.org or by contacting ACM Media Sales at (212) 626-0654.

Single Copies
Single copies of *Communications of the ACM* are available for purchase. Please contact acmhelp@acm.org.

COMMUNICATIONS OF THE ACM

(ISSN 0001-0782) is published monthly by ACM Media, 2 Penn Plaza, Suite 701, New York, NY 10121-0701. Periodicals postage paid at New York, NY 10001, and other mailing offices.

POSTMASTER

Please send address changes to
Communications of the ACM
2 Penn Plaza, Suite 701
New York, NY 10121-0701 USA

Association for Computing Machinery

Printed in the U.S.A.

DOI:10.1145/1506409.1506410

Moshe Y. Vardi

Conferences vs. Journals in Computing Research

An old joke tells of a driver, returning home from a party where he had one drink too many, who hears a warning over the radio about a car careening down the wrong side of the highway. "A car?" he wondered aloud,

"There are lots of cars on the wrong side of the road!"

I am afraid that driver is us, the computing-research community. What I'm referring to is the way we go about publishing our research results. As far as I know, we are the only scientific community that considers conference publication as the primary means of publishing our research results. In contrast, the prevailing academic standard of "publish" is "publish in archival journals." Why are we the only discipline driving on the conference side of the "publication road?"

Conference publication has had a dominant presence in computing research since the early 1980s. Still, during the 1980s and 1990s, there was ambivalence in the community, partly due to pressure from promotion and tenure committees about conference vs. journal publication. Then, in 1999, the Computing Research Association published a Best Practices Memo, titled "Evaluating Computer Scientists and Engineers for Promotion and Tenure," that legitimized conference publication as the primary means of publication in computer research. Since then, the dominance of conference publication over journals has increased, though the ambivalence has not completely disappeared. (In fact, ACM publishes 36 technical journals.)

Recently, our community has begun voicing discomfort with conference publication. A Usenix Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems (WOWCS), held in San Francisco in April 2008, focused on the paper selection process, which is not working too well these days, according to many people. (You can find the proceedings at http://www.usenix.net/events/wowcs08/ and a follow-up wiki at http://wiki.usenix.org/bin/view/Main/Conference/CollectedWisdom.)

Two presentations at the workshop evolved into thought-provoking *Communications*' Viewpoint columns. In the January 2009 issue, we published "Scaling the Academic Publication Process to Internet Scale" by J. Crowcroft, S. Keshav, and N. McKeown (p. 27). In this issue, you will find "Program Committee Overload in Systems" by K. Birman and F.B. Schneider (p. 34). The former attempts to offer a technical solution to the paper-selection problem, while the latter points us to the nontechnical origins of the problem, expressing hope to "to initiate an informed debate and a community response."

I hope the outcome from WOWCS and the Viewpoint columns published here will initiate an informed debate. But I fear these efforts have not addressed the most fundamental question: Is the conference-publication "system" serving us well *today*? Before we try to fix the conference publication system, we must determine whether it is worth fixing.

My concern is our system has compromised one of the cornerstones of scientific publication—peer review. Some call computing-research conferences "refereed conferences," but we all know this is just an attempt to mollify promotion and tenure committees. The reviewing process performed by program committees is done under extreme time and workload pressures, and it does not rise to the level of careful refereeing. There is some expectation that conference papers will be followed up by journal papers, where careful refereeing will ultimately take place. In truth, only a small fraction of conference papers are followed up by journal papers.

Years ago, I was told that the rationale behind conference publication is that it ensures fast dissemination, but physicists ensure fast dissemination by depositing preprints at www.arxiv.org and by having a very fast review cycle. For example, a submission to *Science*, a premier scientific journal, typically reaches an editorial decision in two months. This is faster than our conference publication cycle!

So, I want to raise the question whether "we are driving on the wrong side of the publication road." I believe that our community must have a broad and frank conversation on this topic. This discussion began in earnest in a workshop at the 2008 Snowbird Conference on "Paper and Proposal Reviews: Is the Process Flawed?" (see http://doi.acm.org/10.1145/1462571.1462581).

I cannot think of a forum better than *Communications* in which to continue this conversation. I am looking forward to your opinions.

Moshe Y. Vardi, EDITOR-IN-CHIEF

ACM's Online Books & Courses Programs!

Helping Members Meet Today's Career Challenges

NEW! Over 2,500 Online Courses in Multiple Languages Plus 1,000 Virtual Labs from Element K!

element k
your learning solutions partner

ACM's new Online Course Collection includes over **2,500 online courses in multiple languages, 1,000 virtual labs, e-reference tools, and offline capability**. Program highlights:

The ACM E-Learning Catalog - round-the-clock access to 2,500+ online courses on a wide range of computing and business topics, in multiple languages.

Exclusive vLab® Virtual Labs - 1,000 unique vLab® exercises place users on systems using real hardware and software allowing them to gain important job-related experience.

Reference Tools - an e-Reference Library extends technical knowledge outside of the classroom, plus online Executive Summaries and quick reference cards to answer on-the-job questions instantly.

Offline Player - members can access assessments and self-study courses offline, anywhere and anytime, without a live Internet connection.

A downloadable Quick Reference Guide and a 15-minute site orientation course for new users are also available to help members get started.

The ACM Online Course Program is open to ACM Professional and Student Members.

---

600 Online Books from Safari

ACM members are eligible for a **special 40% savings** offer to upgrade to a Premium or Full Library subscription through June 15, 2009. For more details visit: http://pd.acm.org/books/about_sel.cfm

The ACM Online Books Collection includes **full access to 600 online books** from Safari® Books Online, featuring leading publishers including O'Reilly. Safari puts a complete IT and business e-reference library right on your desktop. Available to ACM Professional Members, Safari will help you zero in on exactly the information you need, right when you need it.

Safari® Books Online

500 Online Books from Books24x7

All Professional and Student Members also have **full access to 500 online books** from Books24x7®, in ACM's rotating collection of complete unabridged books on the hottest computing topics. This virtual library puts information at your fingertips. Search, bookmark, or read cover-to-cover. Your bookshelf allows for quick retrieval and bookmarks let you easily return to specific places in a book.

books24x7® a SkillSoft Company

Association for Computing Machinery

Advancing Computing as a Science & Profession

pd.acm.org
www.acm.org/join

DOI:10.1145/1506409.1506412

Logic of Lemmings in Compiler Innovation

I AM DEEPLY ambivalent about what I read in the contributed article "Compiler Research: The Next 50 Years" by Mary Hall et al. (Feb. 2009). On the one hand, its description of the field's challenges and opportunities evoke great excitement; on the other, the realities cast a discouraging pall on that excitement.

The practical adoption of useful research results is generally a slow process, taking up to a decade or more to achieve. In compilers, however, technology transfer has actually proceeded negatively.

It has been at least four decades since the idea first emerged that, besides translating to machine code, a compiler must be able to perform a second important function: automate detection of a large class of programming errors without the need for massive test suites. What followed was a series of programming languages and their compilers embodying this idea that at first (1970s and 1980s) software practitioners began to adopt at a typical rate.

But in the following decade, the industry reversed course, choosing C and later C++, which not only allow, but routinely require, highly unsafe methods scarcely above the assembly-language level, with huge regions of semantics that are explicitly disavowed as "undefined." Academic researchers and educators resisted this reversal for another decade, reasoning that safe languages would teach better habits, improve unsafe languages, and be all the more important when using unsafe languages. Eventually, however, they also succumbed to intense pressure and acquiesced to their role as industry minion.

Advocating for better language and compiler technology, I have almost never been rebutted by an argument beyond "It's what everybody is doing." It seems that the logic of lemmings is the only persuasive reasoning in the area.

The trend has now shifted toward pervasive use of scripting languages that abandon static safety altogether. The result is that developing large test suites is the only significant, viable means of ensuring quality and security. This has happened at the same time Internet attacks and concurrency have made these very qualities much more important. It is perhaps a difficult call whether better dynamic safety but worse static safety is good or bad but is certainly not a step forward.

The unpleasant truth is that almost the entire software community has resoundingly rejected the best research in compilers and languages, despite being well-proven as eminently practical for decades. Unless someone finds a way to dramatically change the attitudes of software developers, much of the exciting work Hall et al. envision for the next 50 years will be relegated to the role of academic exercise, as has happened for the past 40.

Rodney M. Bates, Wichita, KS

Authors' Response:

We agree with Bates that the software industry has been slow to adopt research ideas invented by the programming-languages and compiler communities.

Nevertheless, tools based on model checking are routinely used to verify Windows device drivers, and Google uses its MapReduce programming model for processing large-scale data sets.

Both model checking and MapReduce are based on research from the programming languages and compiler communities. We anticipate many more successful technology transitions of this sort in the future.

Mary Hall, Salt Lake City, UT
David Padua, Urbana-Champaign, IL
Keshav Pingali, Austin, TX

To Attract Women to Computer Science, Stress Love of Learning

I could hardly believe that a review article discussing "Women in Computing" (Feb. 2009) would quote a woman saying the best advice she received about how women should compete in the workplace is to "Look like a girl. Act like a lady. Think like a man. Work like a dog" (Jean Bartik, programmer for the Eniac computer). What precisely does each sentence mean? The whole statement sounds sexist to me, to say the least.

I deeply disagree with Caitlin Kelleher's statement "If we want young girls to choose to learn how to program computers, we need to deeply understand the kinds of programs girls will be motivated to create and design programming environments that make those programs readily achievable." This, too, makes no sense. Science is science, and the main motivation for doing science is the learning itself and the inner satisfaction and understanding knowledge delivers. If women cannot be motivated by learning and knowledge, they should not be doing science.

More important than making computing something that would please women so as to attract them is to educate them about the importance of science and knowledge and the inherent satisfaction they can bring any person.

Many of the "strategies" described as successful for the recruitment and retention of women in computing are, in my view, ways of reinforcing the existing bias against women in science (such as redesigning introductory CS courses to emphasize applications in areas of interest to women). This would succeed only at a superficial level, turning women into, perhaps, competent CS users.

Maria do Carmo Nicoletti,
São Paulo, Brazil

To Learn Software Engineering, Study Application Logic

The question posed by the "The Profession of IT" Viewpoint "Is Software Engineering Engineering?" (Mar. 2009) by Peter J. Denning and Richard D. Riehle was much too narrow. The fact is that most of us aren't math-

ematicians, scientists, or engineers but rather accountants. I am a case in point, having spent half of my career working either directly on accounting/business applications or on the operating-system kernels underlying database servers.

Although the production of computer software does not typically resemble anything a mathematician would endorse or condone, it is nevertheless analogous to mathematics in that it serves as handmaiden to science, engineering, business, entertainment, and sometimes even mathematics. Therefore, the relationship between software engineering and the traditional engineering disciplines depends on which of these masters it happens to be serving, in other words, its context.

Paul E. McKenney, Beaverton, OR

Authors' Response:

Rather than take on the whole of software development, we restricted ourselves to whether software engineering is genuine engineering. Behind our question is the frequent sniping from other engineering fields that CS graduates cannot do basic engineering things (such as predict the failure modes of software and their attendant risks).

It is an interesting question whether augmenting software engineering with other aspects of software development would make it more like engineering. We doubt it would, but it is a great topic for a future column.

Peter J. Denning and Richard D. Riehle, Monterey, CA

Praise for the GT.M Database Engine

In his article "Parallel Programming with Transaction Memory" (Feb. 2009), Ulrich Drepper said that although transactions are familiar to database developers, their packaging is unfamiliar to systems programmers. Although he views software transactional memory (STM) as current research, the fact is that STM (embodied in the GT.M database engine, fis-gtm.com) is mature, proven technology in daily production use. GT.M provides so-called ACID, or atomic, consistent, isolated, and durable, transactions but in a schema-less database engine packaged as scalar and multidimensional associative memory (arrays) familiar to systems programmers. As the platform for the Fidelity Information Services Profile banking application (fis-profile), GT.M has been available for years, notably in banking and finance (tens of millions of accounts worldwide), running one of the largest, if not the largest, real-time core processing system at any bank anywhere in the world (tinyurl.com/asmque).

Drepper's sample function `f1 _ 1()` he used to illustrate STM could be coded in GT.M in a procedural style familiar to systems programmers:

```
f11(r,t)
   TStart ()
   Set @t=timestamp1
   Set @r=$Increment(counter1)
   TCommit
   Quit
```

For code bracketed by TStart/TCommit commands, the GT.M runtime system ensures the ACID properties, no matter how many processes execute the code at the same time. At TCommit, if no variables accessed by the transaction have changed since TStart, the runtime system commits the updates. If one or more variables has changed, the code automatically restarts from Tstart. Except for a small critical section internal to the runtime during TCommit, the processes run in parallel; to prevent "live locks" in the event the updates cannot be committed on the third try, the entire transaction is executed within a critical section on the fourth try. In the SMP multicore environments on which we benchmark Profile/GT.M, we routinely observe linear to near-linear scalability (up to tens of processors/cores and hundreds of concurrent processes).

GT.M includes a compiler and language environment for the M (or MUMPS) language, so M and C are able to call each other, and the top-level program can be a C main(). Since the software is freely available under the AGPL v3 FOSS license (sourceforge.net/projects/fis-gtm), no technical or licensing barriers prevent creation of a preferred API to expose the underlying engine to a C programmer. Also worth noting is that the database engine uses a daemonless architecture and requires only ordinary user and group privileges to run.

GT.M's software transactional memory is a mature, proven technology, though more research is always welcome.

K.S. Bhaskar, Malvern, PA

Crediting SABRE's Sources

The "Economic and Business Decisions" Viewpoint "The Extent of Globalization of Software Innovation" by Ashish Arora et al. (Feb. 2009) referred to "...IBM's SABRE airline reservation..." There is indeed no such entity. SABRE is software developed by American Airlines that runs (in part) on IBM's Transaction Processing Facility operating system. TPF's predecessor, the Airlines Control Program, was developed from work done at American Airlines (and other organizations, including United Airlines) where the reservation system is called APOLLO. So there is a close association between IBM and SABRE, but SABRE is not an IBM product and never has been.

John Schlesinger, London, U.K.

Authors' Response:

Computer industry histories like Martin Campbell-Kelly's From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry, *MIT Press, 2004, show that the SABRE system was a joint IBM-American Airlines project, developing the airline industry's first passenger-name record system. Similar systems were developed for other airlines by IBM. Our use of "IBM's SABRE airline reservation" was informal and consistent with the intent of the paragraph, namely that development of innovative systems typically involves close collaboration with lead users.*

Ashish Arora, Pittsburgh, PA
Matej Drev, Pittsburgh, PA
Chris Forman, Atlanta, GA

Communications welcomes your opinion. To submit a Letter to the Editor, please limit your comments to 500 words or less and send to letters@cacm.acm.org.

BLOG@CACM

The *Communications* Web site, cacm.acm.org, features 13 bloggers in the BLOG@CACM community. In each issue of *Communications*, we'll publish excerpts from some of their posts, plus readers' comments.

DOI:10.1145/1506409.1506434 cacm.acm.org/blogs/blog-cacm

Recommendation Algorithms, Online Privacy, and More

Greg Linden, Jason Hong, Michael Stonebraker, and Mark Guzdial discuss recommendation algorithms, online privacy, scientific databases, and programming in introductory computer science classes.

From Greg Linden's "What is a Good Recommendation Algorithm?"

Netflix is offering one million dollars for a better recommendation engine. Better recommendations clearly are worth a lot.

But what are better recommendations? What do we mean by "better"?

In the Netflix Prize, the meaning of better is quite specific. It is the root mean squared error (RMSE) between the actual ratings Netflix customers gave the movies and the predictions of the algorithm.

Let's say we build a recommender that wins the contest. We reduce the error between our predictions and what people actually will rate by 10% over what Netflix used to be able to do. Is that good?

Depending on what we want, it might be very good. If what we want to do is show people how much they might like a movie, it would be good to be as accurate as possible on every movie.

However, this might not be what we want. Even in a feature that shows people how much they might like any particular movie, people care a lot more about misses at the extremes. For example, it could be much worse to say that you will be lukewarm (a prediction of 3½ stars) on a movie you love (an actual of 4½ stars) than to say you will be slightly less lukewarm (a prediction of 2½ stars) on a movie you are lukewarm about (an actual of 3½ stars).

Moreover, what we often want is not to make a prediction for any movie, but find the best movies. In TopN recommendations, a recommender is trying to pick the best 10 or so items for someone.

A recommender that does a good job predicting across all movies might not do the best job predicting the TopN movies. RMSE equally penalizes errors on movies you do not care about seeing as it does errors on great movies, but perhaps what we really care about is minimizing the error when predicting great movies.

There are parallels here with Web search. Web search engines primarily care about precision (relevant results in the top 10 or top three). They only care about recall when someone would notice something they need missing from the results they are likely to see. Search engines do not care about errors scoring arbitrary documents, just their ability to find the TopN documents.

Aggravating matters further, in both recommender systems and Web search, people's perception of quality is easily influenced by factors other than the items shown. People hate slow Web sites and perceive slowly appearing results to be worse than fast-appearing results. Differences in the information provided about each item, especially missing data or misspellings, can influence perceived quality. Presentation issues, even the color of links, can change how people focus their attention and which recommendations they see. People trust recommendations more when the engine can explain why it made them. People like recommendations that update immediately when new information is available. Diversity is valued; near duplicates disliked. New items attract attention, but people tend to judge unfamiliar or unrecognized recommendations harshly.

In the end, what we want is happy, satisfied users. Will a recommenda-

tion engine that minimizes RMSE make people happy?

Reader's comment:

Another thing that seems to be often overlooked is how you get users to trust recommendations. When I first started playing with recommendation algorithms I was trying to produce novel results—things that the user didn't know about and would be interesting to them, rather than using some of the more basic counting algorithms that are used for Amazon's related products.

What I realized pretty quickly is that even I didn't trust the recommendations. They seemed disconnected, even if upon clicking on them I'd realize they were, in fact, interesting and related.

What I came to from that was that in a set of recommendations you usually want to scale them such that you slip in a couple of obvious results to establish trust—things the user almost certainly knows of, and probably won't click on, but they establish, "OK, yeah, these are my taste." Then you apply a second ranking scheme and jump to things they don't know about. Once you've established trust of the recommendations they're much more likely to follow up on the more novel ones.

—Scott Wheeler

From Jason Hong's "Privacy as... Sharing More Information?"

What I *am* saying is that, rather than just viewing privacy as not sharing information with others, or viewing privacy as projecting a desired persona, we should also consider how to make systems so that people can safely share more information and get the associated benefits from doing so....

There are many dimensions here in this design space. We can change what is shared, how it is shared, when something is shared, and who it is shared with. One key challenge is in balancing privacy, utility, and the overhead for end users in setting up these policies. Another key challenge is understanding how to help people change these policies over time to adapt to people's needs. These are issues I'll discuss in future blog postings.

For me, a particularly intriguing way of thinking here is safe staging, an idea that Alma Whitten brought to the attention of security specialists in her seminal paper *Why Johnny Can't Encrypt*. The basic idea is that people progressively get more powerful tools as they become comfortable with a system, but are kept in a safe state as much as possible as they learn how to use the system. A real-world example would be training wheels on a bicycle. For systems that provide any level of awareness, the defaults might be set, for example, so that at first only close friends and family see anything, while over time people can easily share more information as they understand how the system works and how to control things.

From Michael Stonebraker's "DBMSs for Science Applications: A Possible Solution"

Personally, I believe that there are a collection of planet-threatening problems, such as climate change and ozone depletion, that only scientists are in a position to solve. Hence, the sorry state of DBMS support in particular (and system software support in general) for this class of users is very troubling.

Science users, of course, want a commercial-quality DBMS, i.e., one that is reliable, scalable, and comes with good documentation and support. They also want something that is open source. There is no hope that such a software system can be built in a research lab or university. Such institutions are good at prototypes, but not production software. Hence, the obvious solution is a nonprofit foundation, along the lines of Apache or Mozilla, whose charter would be to build such a DBMS. It could not be financed by venture capital, because of market size issues. As such, support must come from governments and foundations.

It is high time that the United States got behind such an initiative.

Reader's comment:

While I agree that RDBMS is not an optimal technology for scientific applications and that an open source initiative may lead to some good innovation, I'd be cautious in separating the data model from the query and management language.

There are proprietary tools, such as kx.com, that have done so successfully. The speed and capacity of such tools is phenomenal (as are the licensing fees one must pay).

—Leonidas Irakliotis

From Mark Guzdial's "The Importance of Programming in Introductory Computing Courses"

In computer science, the way that we investigate computation is with programming. We don't want to teach computing as a pile of "accumulated knowledge." We know that that doesn't lead to learning. We need to teach computation with exploration and investigation, which implies programming.

The best research study that I know of that addresses this question is Chris Hundhausen's study where he used algorithmic visualization in CS1. He had two groups of students. One group was to create a visualization of an algorithm using art supplies. The students were learning theory and describing the process without programming. The second group had to use a visualization system, ALVIS. The students were learning theory and encoding their understanding in order to create a presentation. As Chris says in his paper, "In fact, our findings suggest that ALVIS actually had a key advantage over art supplies: namely, it focused discussions more intently on algorithm details, leading to the collaborative identification and repair of semantic errors." If you have no computer system, it's all too easy to say "And magic happens here." It's too easy to rely on intuitive understanding, on what we think ought to happen. Having to encode a solution in something that a computer can execute forces an exactness such that errors can be identified.

The idea isn't that programming creates barriers or makes it harder. Rather, using the computer makes it *easier* to learn it *right*. Without a computer, it's *easier* to learn it *wrong*, where you just learn computing as a set of accumulated knowledge (as described in the AAAS report) or with semantic errors (as with art supply algorithm visualization). If you don't use programming in CS1, you avoid tedious detail at the possible (even likely) loss of real learning.

DOI:10.1145/1506409.1506413 David Roman

The Print-Web Partnership Turns the Page

The relationship between *Communications'* Web site and its print forefather is entering a new era this month with the debut of the blog@CACM. In this issue (p. 10), you'll find excerpts from essays published online at http://cacm.acm.org/blogs/blog-cacm, plus some recent online reader comments. The reason we've chosen to publish select blogs each month is simple: *Communications*' expert bloggers write valuable posts and *Communications*' credo is to disseminate valuable information that advances the arts, sciences, and applications of information technology. Readers have noticed the high quality of these blogs, making them, as well as our syndicated blogs (http://cacm.acm.org/blogs), some of the site's most popular sections.

The blog@CACM also gives the online *Communications* a unique bonus: a commenting feature that enables sometimes extensive discussions of industry issues, which is, of course, the beauty of blogs. The back-and-forth exchanges and clarification of blog posts and other site content create a round-the-clock equivalent of the Greek forum.

The magazine's blog pages might change over time as we learn readers' preferences: be they more or fewer posts, shorter or longer excerpts, with or without related comments. For now we're marking the beginning of a productive relationship between print and online outlets.

Exploring the Relevant Past

Clicking through the magazine archive (http://cacm.acm.org/magazines) is a pleasure similar to paging through an old photo album. There are familiar names and familiar topics. Most striking is the prescience and enduring relevance of many articles. Peruse the decades and see the early work of future A.M. Turing Award winners and industry icons. Read about the computer industry's manpower shortage concerns in "U.S. Productivity in Crisis" (June 1981). China's growing prowess is the subject of "Computer Technology in Communist China" in September 1966. Steve Jobs, then with NeXT Inc., describes the importance of user interfaces and user apps in April 1989. And that's just scratching the surface.

The magazine's covers followed their own trends. The blue-and-white period in the 1960s transformed into the stark blue-and-black period in the 1970s, that gave way to full-color illustrations by the 1980s.

There's mystery as well. Why was Miss U.S.A. on the June 1965 cover of *Communications*?

ACM Member News

VELOSO WINS SIGART AWARD

Manuela A. Veloso, a professor of computer science at Carnegie Mellon University, received the 2009 Autonomous Agents Research Award from SIGART. "Professor Veloso's research is particularly noteworthy for its focus on the effective construction of teams of robot agents, where cognition, perception and action are seamlessly integrated to address planning, execution and learning tasks," noted the SIGART award citation.

MYERS RECEIVES SIGPLAN AWARD

Andrew C. Myers, a professor of computer science at Cornell University, won an award for the Most Influential POPL Paper presented at the POPL symposium held 10 years prior to the award year. In its award announcement, the judges noted that Myers' 1999 paper, *JFlow: Practical Mostly-Static Information Flow Control*, "demonstrated the practicality of using static information flow analysis to protect privacy and preserve integrity by giving an efficient information flow type checker for an extension of the widely used Java language."

CONSTANTINE WINS STEVENS AWARD

ACM Fellow Larry Constantine, director of the Laboratory for Usage-Centered Software Engineering at the University of Madeira, is this year's recipient of the Stevens Award. The award, managed by the Reengineering Forum, recognizes "outstanding contributions to the literature or practice of methods for software and systems development."

GRACE HOPPER CELEBRATION OF WOMEN IN COMPUTING

The 9th Annual Grace Hopper Celebration of Women in Computing will take place from September 30 to October 3, 2009 in Tucson, AZ. This year's theme, "Creating Technology for Social Good," recognizes the significant role women play in defining technology used to solve social issues. Scholarship applications are now being accepted; the deadline is May 27.

Science | DOI:10.1145/1506409.1506414

Kirk L. Kroeker

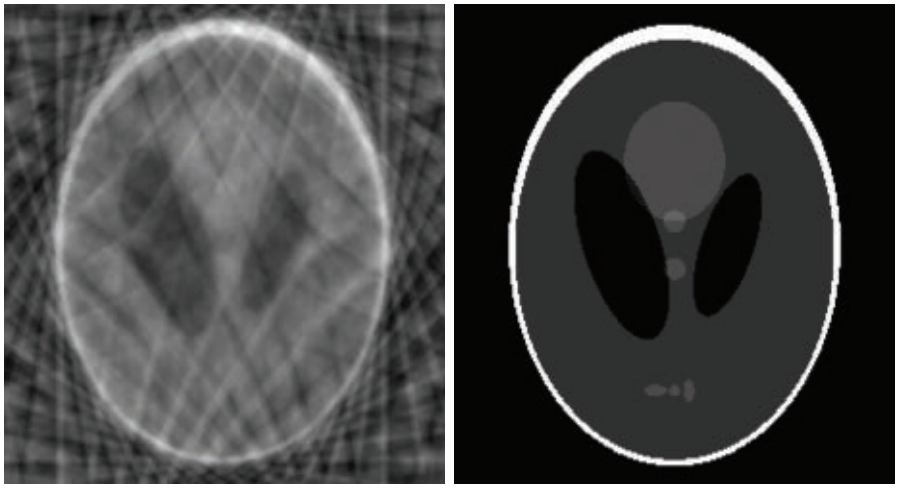
Rethinking Signal Processing

Compressed sensing, which draws on information theory, probability theory, and other fields, has generated a great deal of excitement with its nontraditional approach to signal processing.

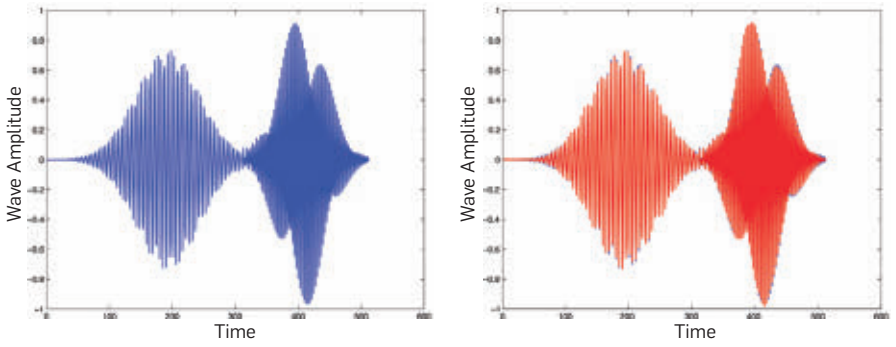
For many years, traditional signal processing has relied on the Shannon-Nyquist theory, which states that the number of samples required to capture a signal must be determined by the signal's bandwidth. An alternative sampling theory, called compressed sensing or compressive sampling, turns the Shannon-Nyquist theory on its head. The idea behind compressed sensing is to accurately acquire signals from relatively few samples. The theory was so revolutionary when it was created a few years ago that an early paper outlining it was initially rejected on the basis that its claims appeared impossible to substantiate.

Today, however, compressed sensing is attracting a great deal of interest from mathematicians, computer scientists, and both optical and electrical engineers. And the theory is inspiring a new wave of lab work to produce systems that require far less power and operate more efficiently than those that rely on the traditional capture-compress paradigm. These systems include applications for industrial imaging, digital photography, biomedical imaging, and other forms of analog-to-digital conversion.

Compressed sensing emerged initially from an experiment inspired by a real-world problem with magnetic resonance imaging (MRI). The goal of the experiment, headed by Charles Mistretta at the University of Wisconsin, Madison, was to speed up the notoriously slow MRI process to make it more comfortable for patients, compensate for their minor movements, increase MRI throughput, and possibly even make the process fast enough to conduct 3D imaging. Because MRI hardware relies on a quantum effect to determine the density of protons in a patient's body, the data-capture process cannot be shortened by improving the hardware's core technology. Therefore, the question initially posed by the researchers working on the problem is whether the time it takes to perform an MRI can be reduced by capturing fewer samples and reconstructing a full image from only a small fraction of the traditional amount of required data. While conventional sampling theory suggests doing so would not be possible, the University of Wisconsin researchers applied standard image-reconstruction algorithms on heavily subsampled MRI data. But the results were inadequate, so the researchers turned to Emmanuel Candes, a professor of applied and computational mathematics at the California Institute of Technology, for as-



The left MRI image suffers from blurred edges, numerous artifacts, and low resolution. The phantom image on the right was produced with minimally sampled Fourier coefficients using 5% of the original MRI data, and is the same as the original MRI phantom (not shown here).



The left image represents high-frequency radar pulses. In the right image, the original signal (blue) is overlapped by the reconstructed signal (red), which was built through compressed sensing at a rate that is 6% of what is required by the Shannon-Nyquist theory.

sistance. Candes and his Caltech team set out to reconstruct the MRI images without any artifacts and by using only 5% of the sampled imaging data.

"When I looked at the artifacts, I discovered that they had certain features that I knew I could make go away by penalizing them in the reconstruction," says Candes, who notes he was simply hoping his algorithm would improve the quality of the images. "That's where the surprise came in," he says. "What I was not expecting was that it would give me the truth." Candes says that he and his team quickly realized they could do something that nobody thought was possible: simultaneous acquisition and compression. "That was the birth of compressed sensing," he says. "We found that you can reconstruct images from dramatically fewer samples than what was previously necessary."

Justin Romberg, who worked with Candes on the initial MRI project, points out that finding sparse signals that satisfy a set of linear constraints was an idea "floating around in the literature" at the time. However, he says, no existing theories supported the notion that it would be possible to perform reconstruction from limited data. "We were the first people to talk about it in this way," says Romberg, a professor of electrical and computer engineering at the Georgia Institute of Technology. Of course, compressed sensing does not make it possible to reconstruct anything and everything from limited information. The target image or data set must have some special structure. "If there is structure, you can actually do much better than the Shannon-Nyquist theorem dictates," says Romberg. "You can sample more efficiently."

There are many projects in research labs around the world to build hardware that can leverage some of the core ideas associated with compressed sensing, so one might assume that the theory has come of age. But given the requirement to know some structure of the expected signal prior to sampling—implying that a random signal or one consisting entirely of noise would not be well suited to compressed sensing—the research team sought to establish firm mathematical foundations for their results. "For the theory, we know a lot today, not all that we would like to know," says Candes. "But in broad strokes, the foundation is there."

Theoretical Applications

One of the people who helped establish this foundation is Terence Tao, a professor of mathematics at the University of California, Los Angeles. "Emmanuel had found a toy problem in pure mathematics which, if solved, could lead to a practical demonstration that compressed sensing could actually work effectively," says Tao. "That problem was in two areas in my own expertise—Fourier analysis and random matrices—and so I started to play around with it." Eventually, says Tao, he, Candes, and Romberg solved that toy problem, establishing that compressed sensing worked for a certain type of measurement related to the Fourier transform, and started working together to further develop the theory. "I would not say that the field is anywhere as mature as, say, Shannon's theory of information, or the statistical theory of least squares regression, which are some of the precursors to this subject," says Tao. "But the core ideas of the subject are by now quite well understood, even if there are still many areas where we would like to develop them further."

Compressed sensing is leading to new ways of looking at math problems in seemingly unrelated areas.

One of the areas that needs more attention, according to Tao, is how the theory is centered around linear measurement. "We don't yet know what to do if our measurement devices behave nonlinearly with respect to the data," Tao says. "We are still exploring exactly what type of measurement models compressed sensing excels at, and where the paradigm reaches its limits and must be replaced or supplemented by a different type of method."

Compressed sensing works for a large number of special-purpose situations, says Tao, but is probably not suitable as a general-purpose tool. For instance, he says, it is unlikely that general-purpose digital cameras will rely on compressed sensing, given that consumers might want to take pictures that look like random, unstructured images. "But a dedicated sensor network that is devoted to detecting a certain special type of signal might benefit substantially from this paradigm," he says.

Indeed, compressed sensing is having an impact on the designs of a broad array of such applications, given that sensors can be found almost everywhere. Engineers at Rice University, for example, are working on a single-pixel camera that can take high-quality photos by using compressed sensing and a digital micromirror array. In addition, space agencies have shown interest in the theory, with initial designs outlined for cameras that rely on compressed sensing to save power in deep space. And Candes and Romberg are working on a project with DARPA to overcome some of the traditional limitations associated with the analog-to-digital conversion of radio signals. The project's goal is to design a system for monitoring radio frequency bands much more

efficiently than is currently possible. The first chip for the project, which will sample frequencies at a rate of 800 million data points per second, is in fabrication now, and should soon be ready for testing. "One application for this kind of system," says Romberg, "would be for monitoring large swaths of communications bandwidth, where you don't necessarily know which frequency would be used for communicating."

Mathematical Insights

In addition to having an impact on the design of sensor systems and other industrial applications, compressed sensing is leading to new ways of looking at math problems in seemingly unrelated areas. Candes and Tao, for example, are currently working on the problem of matrix prediction, the most widely known example of which is the Netflix Prize. The goal of those working to win the prize is to improve the accuracy of the Netflix movie-recommendation system. Each Netflix customer watches and rates a small fraction of movies, so it is possible to know only a little of the matrix in advance. While other mathematical approaches, such as spectral graph theory, have been applied to such matrix-prediction problems, Candes and Tao say there are strong parallels to the kinds of problems that compressed sensing can address. "The point is that we believe the ratings matrix to be structured," says Tao. "Emmanuel and I are not working directly on the Netflix Prize problem, but on some more foundational mathematical issues related to one approach to solving this problem."

Compressed sensing has applications for biomedical imaging, digital photography, and other forms of analog-to-digital conversion.

As for the future of the theory, Romberg says that one challenge remaining for those working on compressed sensing is convincing people that there is some value in it, and a corresponding value in changing sensor systems that have been implemented in certain ways since the beginning of signal processing. "A lot of the theory of compressed sensing," he says, "goes against everything that sensors have been designed to do." Another challenge is developing more efficient reconstruction algorithms. Traditionally, the signal-processing workload happens during encoding (such as for music and image files), while the decoder does very little. In compressed sensing, the workload is reversed; the encoder does very little, but the decoder has to work to find the location of the signal, its amplitude, and other characteristics. "A question that is active and that must remain active is how to get very fast algorithms to do the reconstruction," says Candes.

For his part, Tao says compressed sensing is here to stay. "Perhaps in five or 10 years most of the issues people are actively studying now will be resolved or their limitations understood much better," he says. "There is certainly a lot of potential, particularly in specific fields such as MRI, in which there was a definite need to squeeze more information out of fewer measurements."

But compressed sensing's impact, Tao says, is likely to be uneven, given that traditional methods might be more effective for some applications due to the limitations of compressed sensing that aren't completely understood.

According to Candes, at least one impact of the theory is happening outside the research labs and on a more organic, social level. Candes says that when he attends conferences related to compressed sensing, he regularly sees pure mathematicians, applied mathematicians, computer scientists, and hardware engineers coming together to share ideas about the theory and its applications. "It's really exciting to see all these people talk together," Candes says. "I know compressed sensing is changing the way people think about data acquisition." C

Based in Los Angeles, **Kirk L. Kroeker** is a freelance editor and writer specializing in science and technology.

Obituary

Jacob T. Schwartz, 79, Dies

Jacob T. "Jack" Schwartz, a mathematician and computer scientist who conducted important research in a wide variety of fields and founded the department of computer science at New York University, died on March 2. He was 79.

Schwartz was well respected by his peers for his brilliance as a scientist, his skill and vision as a department chair, and a seemingly boundless intellectual curiosity. He first made a name for himself as a mathematics graduate student at Yale when he co-authored, with his Ph.D. advisor Nelson Dunford, the three-volume *Linear Operators*. The text was first published in 1958 and, a half-century later, is still in print. (Dunford and Schwartz were jointly awarded the Leroy P. Steele Prize from the American Mathematical Society for *Linear Operators* in 1981.)

Among Schwartz's many achievements was pioneering work in optimizing compilers at IBM, with John Cocke and Frances E. Allen, as a visiting scientist; the development of SETL, an early programming language, and the Ultracomputer, one of the first parallel computers; and the authorship of 18 books and more than 100 papers and reports.

Schwartz was chair of the department of computer science at New York University's Courant Institute of Mathematical Sciences from 1964 to 1980, which thrived during and after his term as chair. A fellow professor, Edmond Schonberg, recalls how "in the early 1980s, Jack attended a conference on robotics in Washington, D.C., and when he returned, he said, 'This is a subject full of interesting scientific questions—and it is eminently fundable.' " As a result, the department launched a large-scale robotics effort.

During his time at NYU, Schwartz taught nearly every class offered by the department of computer science. "When Jack got interested in a subject, he would teach a course on it," says Schonberg. "As the course evolved, he would reinvent the subject for himself and define his own approach to it. And when he came to class, he would be ecstatic about having discovered something new, and this was contagious."

Technology | DOI:10.1145/1506409.1506415 David Essex

Matchmaker, Matchmaker

Computational advertising seeks to place the best ad in the best context before the right customer.

Andrei Broder, vice president for computational advertising at Yahoo! Research, presenting a tutorial on Web search and advertising at the 30th Annual International ACM SIGIR Conference in Amsterdam.

THE RAPIDLY CHANGING advertisements that appear on Web pages are often chosen by sophisticated algorithms that match ad keywords to words on a Web page. Take the Chevy ad, for example, that frequently appears on your favorite news site. A real-time ad network at one of the major search engines—Google, MSN, and Yahoo!—might place it on a page of automotive news. But what if the news page's featured article is about a tragic accident caused by a mechanical failure in a Chevy SUV? That's not a page General Motors wants to be associated with, let alone pay good money to advertise on.

Costly mishaps like this could be avoided by a new discipline called computational advertising, which seeks to put the best ad in the best context before the right customer. It draws from numerous fields, including information retrieval, machine learning, natural-language processing, microeconomics, and game theory, and tries to match ads with a variety of user scenarios, such as querying a search engine, reading a Web page, watching a video on YouTube, or instant messaging a friend.

Computational advertising could spur the Web's growth as a medium of mass customization. Better ad matching could quicken the trend toward personalization, making highly specialized magazines, Web sites, and TV channels more financially viable. "Advertising has been the engine that has powered the huge development of the Web," says Andrei Broder, fellow and vice president for computational advertising at Yahoo! Research. "Without advertising, you would not have blogs and search engines."

Computational advertising is a type of automation that tries to replicate what humans might do if they had the time to read Web pages to discern their content and find relevant ads among the millions available. "In the old world of advertising, they deal with few choices and large amounts of money for each choice," Broder says. "We deal with maybe a hundred million potential ads, each worth a fraction of a cent."

A Perfect Match

There are basically three kinds of Web ads. Sponsored search ads are matched to the results of search engine queries; banner ads target particular demographics and venues, typically without regard to a page's content; and contextual advertising, also called context match, applies to other types of Web pages, such as the home page of a financial news site. Computational advertising addresses all three types of ads.

Google, MSN, and Yahoo! use electronic auctions to assign ads to their own results pages and the pages of other Web sites. "Google is a yenta," or matchmaker, says Google chief economist Hal Varian. "The goal is to get a perfect match."

In sponsored search, advertisers bid to place ads that contain keywords correlated to words in a user's search string. For contextual advertising, the keywords are related to words on the entire page, and the search engine's advertising service places the ads. For banner ads, online ad networks place ads on sites whose topics and audiences match the advertiser's criteria.

Before the advent of computational advertising, ad engines could make mistakes more simple-minded than the Chevy SUV scenario. Suppose, for example, a news page contains the word "flowers." If the article isn't about flowers but instead revisits the Rolling Stones' underrated 1967 record *Flowers*, the reader is unlikely to want ads from florists. The old method of analyzing co-occurring words and phrases doesn't help much, and neither does frequency. "You could extract a word used many times in the ar-

PHOTOGRAPH BY NIR NUSSBAUM

ticle and it still is not what the article is about," Broder says.

Therefore, Broder and the 30 researchers who work for him are finding ways to glean the meaning of a page. One promising avenue combines semantic and syntactic features. A semantic phrase categorizes the page and the ads into a 6,000-node topic taxonomy and compares the proximity of the two types of classes as a factor in ranking ads. The hierarchical taxonomy also improves the matching of ads that don't fit a page's exact topic. Keyword matching is still needed to capture more granular content, such as a specific brand of automobile. "We decided that what the article is about should count for about 80% and the words should count for 20%," Broder says.

Another area of interest is using statistical analysis to measure the effect of exogenous events on browsing behavior and adjust the advertisements accordingly. Varian cites short-lived examples, such as this year's rare snowfall in England, or longer-term ones such as the worldwide recession. "In the last few months, there is a big increase in interest in price-sensitive products," Varian says. "The advertisers, in turn, are trying to respond."

All three companies are close-lipped about which of their research has been commercialized, but say that new ideas for algorithms are quickly incorporated into their bidding mechanisms and advertiser tools. Bottom-line results are secret, but the search engines all collect metrics such as revenue per search.

Machine learning, another major focus, concentrates on training algorithms to scan pages for meaning, a technique employed successfully on single-topic documents with the aid of machine-generated labels, but trickier to perform on Web pages, with their assortment of graphics, text, and topics. Microsoft researchers have learned how to employ a type of multiple instance learning to automate classification of sub-documents on pages with incomplete labels and to detect the presence of certain types of content.

"Most of what we do can be boiled down to understanding intent," says Eric Brill, general manager of Microsoft adCenter Labs. By analyzing search strings, for example, algorithms can predict if a person is interested in ads. Some strings are pure attempts at finding information, while others, such as "buy Canon digital camera," have clear commercial intent. "When consumers don't have commercial intent, you don't want to put ads in front of them," Brill says.

Much work focuses on ensuring that new bidding mechanisms don't have incentives for advertisers to misrepresent click-through rates to get better ad placement. In the decentralized economy of the Internet, truthfulness is a currency reinforced by carefully crafted algorithms. "People are out there to make money," says Thore Graepel, a senior researcher at Microsoft Research. "We need to build mechanisms where everyone benefits."

One might expect the speed and volume of data to create a capacity problem, but the researchers express mixed opinions. Graepel says semantic analysis creates an extra burden. "You will hit a computational bottleneck, that's pretty clear," he says. To avoid this, researchers optimize algorithms to make the best decisions with the smallest possible data sets. But they also have faith in engineers' ability to exploit techniques such as parallel processing. "It's surprising how they are always able to scale to deal with these new algorithms," Varian says.

Privacy regulations remain an obstacle to personalizing ads, says Graepel. The existing opt-in, opt-out model lets users choose to reveal personal data in exchange for discounts and other incentives. Researchers are also investigating aggregating data on Web traffic to more accurately match ad categories with coarsely defined groups of users who identify their interests simply by visiting certain types of Web sites.

Fortunately, there is hope for avoiding embarrassments like the ill-placed Chevy ad. Researchers at Microsoft adCenter Labs claim their sub-document classification methods can prevent incompatible ads and Web sites from ever hooking up. You might call it a reverse matchmaker, just the sort of odd little entity the Internet's inventors might never have imagined. C

David Essex is a freelance science writer based in Peterborough, NH.

Education

Computer Science Enrollment Increases

Enrollment in computer science classes in the United States has increased for the first time in six years, according to the Computing Research Association's (CRA's) annual Taulbee Survey.

Total enrollment by majors and pre-majors in computer science is up 6.2% per department over last year. If only majors are considered, the increase is 8.1%, according to the CRA survey, which collected enrollment data in fall 2008 from computer science and computer engineering departments at 192 Ph.D.-granting universities.

"The upward surge of student interest is real and bigger than anyone expected," says Peter Lee, incoming chair of CRA. "The fact that computer science graduates usually find themselves in high-paying jobs accounts for part of the reversal. Increasingly students also are attracted to the intellectual depth and societal benefits of computing technology."

Computer science graduates on average earn 13% more than the average college graduate, according to the U.S. Department of Labor, and future job prospects for computer science graduates are higher than for any other science or engineering field.

The average number of new students per department majoring in computer science is up 9.5% over last year. Computer science departments are replenishing the freshman and sophomore ranks with larger groups than they are graduating as seniors, and computer science graduation rates should increase in two to four years as these new students graduate.

The total number of Ph.D. graduates among responding departments grew to 1,877 for the period July 2007 to June 2008, a 5.7% increase over the previous year.

One area that didn't show improvement is the number of women pursuing computer science degrees, which held steady at 11.8%.

Society | DOI:10.1145/1506409.1506416 Samuel Greengard

Learning Goes Global

In a world that's increasingly global and interconnected, international education is growing, changing, and evolving.

International education isn't exactly a new concept. For years, students have traveled abroad for exchange programs and to obtain degrees. "For many, attending a university in another country is viewed as an ideal way to gain exposure to another culture, learn a language, and participate in an interesting and enriching experience," explains Peggy Blumenthal, chief operating officer for the Institute of International Education in New York City. "It's an important part of the academic environment."

However, in a world that's increasingly global and interconnected, international education is growing, changing, and evolving. Overall, more than 1.5 million students a year study at schools outside their country's borders. According to the Institute of International Education, 173,122 new students enrolled in undergraduate, graduate, and non-degree programs worldwide in 2008—an increase of 7% over the previous year. At the same time, the number of U.S. students studying abroad grew by about 8% to a total of more than 241,791. Some places, such as China, are now experiencing double-digit growth rates.

It's certainly not your mom and dad's summer abroad. What's more, a growing number of these students are from fields such as mathematics, computer science, and natural sciences. "The nature and types of programs are expanding. We're seeing everything from short-term programs that are eight weeks or less to master's programs with a full term abroad," states Brian Whalen, president and CEO of the Forum on Education Abroad and associate provost at Dickinson College in Carlisle, PA. "Technology and communication are changing the way people think about education and making international studies more accessible and popular."

Students learn about studying abroad at the University of Wisconsin, Platteville's International Programs Fair.

PHOTOGRAPH BY ANDY McNEILL

Making the Grade

Study abroad programs once centered mostly on sketching pictures of the Eiffel Tower or learning the finer points of Italian art or German literature. Students in disciplines such as mathematics, computer science, or engineering usually found it difficult, if not impossible, to leave their home institution's program without risking falling behind or veering off track. What's more, most universities weren't inclined to develop exchange programs for those majoring in the sciences.

The situation is changing, however. Thanks to computers, the Internet, and other communication and collaboration tools, the ability to link people and course content is entirely viable. Email, social networking applications such as Facebook, and low- or no-cost calling services such as Skype make it possible for international students to stay in touch with family and friends. In addition, technology and collaboration software—as well as ultra-high-speed Internet2—have made it possible for schools to link programs to one another and create a seamless learning experience. Increasingly, these programs include master's degrees and doctorate degrees.

Hochschule Darmstadt University of Applied Sciences in Germany is among the schools that have jumped onto the international stud-

ies platform. The institution, which serves 11,000 students, commenced its Joint International Master program for computer sciences in 2003. The school partners with the University of Wisconsin, Platteville and James Cook University in Townsville, Australia. At any given time a dozen or so students from these schools venture abroad to study for half-a-year at the partner school. At Hochschule Darmstadt, master's level instruction is entirely in English and graduates receive a joint degree.

"The program provides students with a global perspective and helps them become more attractive on the international job market," says Lucia Koch, director of the International Office for Hochschule Darmstadt. "It also raises the visibility of the school and makes it more attractive and respected."

Koch believes that students who participate in the program gain knowledge and expertise that isn't available in a conventional classroom. "They gain a perspective that can help them understand the field and their future profession better."

Nearly 4,400 miles away in Platteville, WI, Richard D. Shultz, dean of the College of Engineering, Mathematics and Sciences, is reaping benefits as well. A decade ago the school formed a partnership with Hochschule Darmstadt at the undergraduate level. It allowed students from both schools to participate in a conventional exchange program. The relationship evolved after Hochschule Darmstadt suggested expanding the exchange to include its master's program. "It made sense to have a degree that helps students become a citizen of the world," Shultz says. "Students learn different perspectives and discover how people research and work in different parts of the world."

Megan Brenn-White, executive director of the Hessen Universities Consortium, which represents Hochschule Darmstadt and 10 other schools in Germany, believes that an increasingly competitive recruiting environment and a shrinking globe will continue to boost international studies. "Schools are looking to become world-class institutions or boost their stature in the research arena. They're also looking to attract international students for full degree programs because it's often more profitable."

Schools are increasingly developing joint curriculum and collaborating on courses, particularly in computer science and engineering.

Setting a Course

Not surprisingly, the growth of international studies has opened up an entire world of opportunities. Chinese or Argentine students may travel to Germany to receive advanced instruction in mathematics; American or Russian students may venture to New Zealand to receive an education in volcanology. As increasing numbers of schools introduce joint programs—and many institutions turn to U.S. accreditation organizations to gain international acceptance and stature—the playing field is leveling out.

Schools in English-speaking countries, including England, Scotland, Ireland, and Australia, are increasingly the beneficiaries of the trend toward international education. Many of these schools offer outstanding programs at a lower price than students would pay back home.

For example, at the University of Limerick in Ireland, Liam O'Dochartaigh, director of international education, has witnessed an enormous transformation over the last decade. The University of Limerick now has 1,283 students attending from abroad, including about 400 students from the U.S. It also boasts 259 of its own students attending classrooms abroad. The number of international students has spiked more than 100% from a decade ago, he says, and approximately 10% of the student population (the school has approximately 12,500 students) now comes from outside Ireland.

"Universities realize that international study and accessibility is important for financial reasons as well as for international standing," O'Dochartaigh says. He points out that universities are increasingly internationalizing curriculum and schools in different countries even collaborate on coursework and content. The University of Limerick currently has partnerships with 24 schools in Europe and 15 schools in the U.S. and Canada. Tuition derived from international students supplements state funding sources, O'Dochartaigh notes. One foreign student can bring in more than €12,000 per year.

Government organizations are promoting international education programs as well. In the U.S., the National Science Foundation (NSF) has matched more than 2,000 students with intensive eight-week science study grants under its East Asia and Pacific Summer Institutes program since 1990. "There has long been a large interest in students coming to the U.S. to study and do research," says Jong-on Hahm, program manager for the NSF. "But there's a lot of very interesting research that goes on in other countries and American students now have access to it."

The march toward international education will undoubtedly continue. Fueling the trend is the adoption of international standards and the ability to put credits to work at home. In Europe, for example, the Bologna Process—which links ministries, higher-education institutions, students, and staff from 46 countries—guarantees that students receive credits for time spent studying abroad. In addition, schools are increasingly developing joint curriculum and collaborating on courses and studies—particularly in the computer science, engineering, and natural sciences arena.

To be sure, this brave new world of education is creating new vistas. "The educational boundaries between countries are disappearing," says Whalen of Dickinson College. "Students and schools are recognizing that there is a world far beyond their local campus. They're learning that studying aboard presents tremendous opportunities—and advantages." C

Samuel Greengard is an author and freelance writer based in West Linn, OR.

Previous A.M. Turing Award Recipients

1966 A.J. Perlis
1967 Maurice Wilkes
1968 R.W. Hamming
1969 Marvin Minsky
1970 J.H. Wilkinson
1971 John McCarthy
1972 E.W. Dijkstra
1973 Charles Bachman
1974 Donald Knuth
1975 Allen Newell
1975 Herbert Simon
1976 Michael Rabin
1976 Dana Scott
1977 John Backus
1978 Robert Floyd
1979 Kenneth Iverson
1980 C.A.R Hoare
1981 Edgar Codd
1982 Stephen Cook
1983 Ken Thompson
1983 Dennis Ritchie
1984 Niklaus Wirth
1985 Richard Karp
1986 John Hopcroft
1986 Robert Tarjan
1987 John Cocke
1988 Ivan Sutherland
1989 William Kahan
1990 Fernando Corbató
1991 Robin Milner
1992 Butler Lampson
1993 Juris Hartmanis
1993 Richard Stearns
1994 Edward Feigenbaum
1994 Raj Reddy
1995 Manuel Blum
1996 Amir Pnueli
1997 Douglas Engelbart
1998 James Gray
1999 Frederick Brooks
2000 Andrew Yao
2001 Ole-Johan Dahl
2001 Kristen Nygaard
2002 Leonard Adleman
2002 Ronald Rivest
2002 Adi Shamir
2003 Alan Kay
2004 Vinton Cerf
2004 Robert Kahn
2005 Peter Naur
2006 Frances E. Allen
2007 Edmund Clarke
2007 E. Allen Emerson
2007 Joseph Sifakis
2008 Barbara Liskov

Additional information on the past recipients of the A.M. Turing Award is available on: http://awards.acm.org/homepage.cfm?awd=140

ACM A.M. TURING AWARD NOMINATIONS SOLICITED

Nominations are invited for the 2009 ACM A.M. Turing Award. This, ACM's oldest and most prestigious award, is presented for contributions of a technical nature to the computing community. Although the long-term influences of the nominee's work are taken into consideration, there should be a particular outstanding technical achievement that constitutes the principal claim to the award. The award carries a prize of $250,000 and the recipient is expected to present an address that will be published in an ACM journal. Financial support of the Turing Award is provided by the Intel Corporation and Google Inc.

Nominations should include:

1) A curriculum vitae, listing publications, patents, honors, other awards, etc.

2) A letter from the principal nominator, which describes the work of the nominee, and draws particular attention to the contribution which is seen as meriting the award.

3) Supporting letters from at least three endorsers. The letters should not all be from colleagues or co-workers who are closely associated with the nominee, and preferably should come from individuals at more than one organization. Successful Turing Award nominations usually include substantive letters of support from a group of prominent individuals broadly representative of the candidate's field.

For additional information on ACM's award program please visit: www.acm.org/awards/

Nominations should be sent electronically by November 30, 2009 to: Alan Kay, turing@vpri.org

Association for Computing Machinery

Milestones | DOI:10.1145/1506409.1506433

Liskov Wins Turing Award

MIT's Barbara Liskov is the 55th person, and the second woman, to win the ACM A.M. Turing Award.

AWARDS WERE RECENTLY announced by ACM, the British Computer Society, the Computing Research Association, the International Society for Computational Biology, and the National Science Foundation honoring innovative researchers for their contributions to the fields of engineering and computer science.

ACM A.M. Turing Award

Association for Computing Machinery

Barbara Liskov, a professor of engineering and computer science at the Massachusetts Institute of Technology, is the winner of the 2008 ACM A.M. Turing Award. Liskov was cited for her foundational innovations to designing and building the pervasive computer system designs that power daily life. Her achievements in programming language design have made software more reliable and easier to maintain. They are now the basis of every important programming language since 1975, including Ada, C++, Java, and C#.

Previously, computer programs were composed of strings of numbers and characters, but Liskov's work led to the development of object-oriented programming, now the most widespread approach to software development. "Her elegant solutions have enriched the research community, but they have also had a practical effect as well," says ACM president Wendy Hall. "They have led to the design and construction of real products that are more reliable than were believed practical not long ago. In addition to her design features, she focused on engineering innovations that changed the way people thought about programming languages and building complex software. These accomplishments were instrumental in moving concepts out of academia and into the real world."

The Turing Award, widely considered the Nobel Prize in computing, is named for the British mathematician Alan M. Turing. The award carries a $250,000 prize, with financial support provided by Intel Corporation and Google Inc.

PHOTOGRAPH BY DONNA COVENEY/MIT

Lovelace Medal

British Computer Society

Yorick Wilks, a professor of artificial intelligence at Sheffield University, won the Lovelace Medal for his pioneering work on developing virtual agents to assist older people. "I am delighted the BCS is able to recognize the outstanding and sustained contribution Professor Wilks has made during his career to the subject of AI through such a prestigious award," says BCS chief executive David Clarke. "The increasing complexity of the Web will have a profound impact on the way everyone, including the elderly, will live in the future and his work will have a lasting impact on society."

Roger Needham Award

British Computer Society

Byron Cook, a researcher at Microsoft Research at Cambridge University and a professor of computer science at Queen Mary, University of London, won the Needham Award for his creation of TERMINATOR, the first practical tool for automatically proving termination of real-world, imperative programs. "TERMINATOR caused a major stir in the program verification research community when it appeared because it extended Alan Turing's statement on the halting of programs," according to BCS's award announcement. "It has rapidly spilled beyond research circles to the point where TERMINATOR is to be productized by the Windows kernel team."

Distinguished Service Award

Computing Research Association

Eugene Spafford, executive director of CERIAS at Purdue University, won the 2009 Distinguished Service Award in honor of his being "an effective and tireless advocate for the cause of information security research," noted the Computing Research Association in its announcement. "He has been instrumental in keeping public attention on this important research area."

Senior Scientist Award

International Society for Computational Biology

A professor of computer science at Pennsylvania State University, Webb Miller won the Senior Scientist Award for his extensive research in vertebrate genome sequencing.

Overton Prize

International Society for Computational Biology

Trey Ideker, a professor of bioengineering at the University of California, San Diego, who has developed several influential bioinformatics methods and resources, received the Overton Prize as "a scientist in early- or mid-career who has already made a significant contribution to computational biology."

Vannevar Bush Award

National Science Foundation

Millie Dresselhaus, a professor of physics and electrical engineering at the Massachusetts Institute of Technology, was honored with the Vannevar Bush Award for "for her leadership through public service in science and engineering, her perseverance and advocacy in increasing opportunities for women in science, and for her extraordinary contributions in the field of condensed-matter physics and nanoscience."

DOI:10.1145/1506409.1506418

Pierre Larouche

Law and Technology
The Network Neutrality Debate Hits Europe

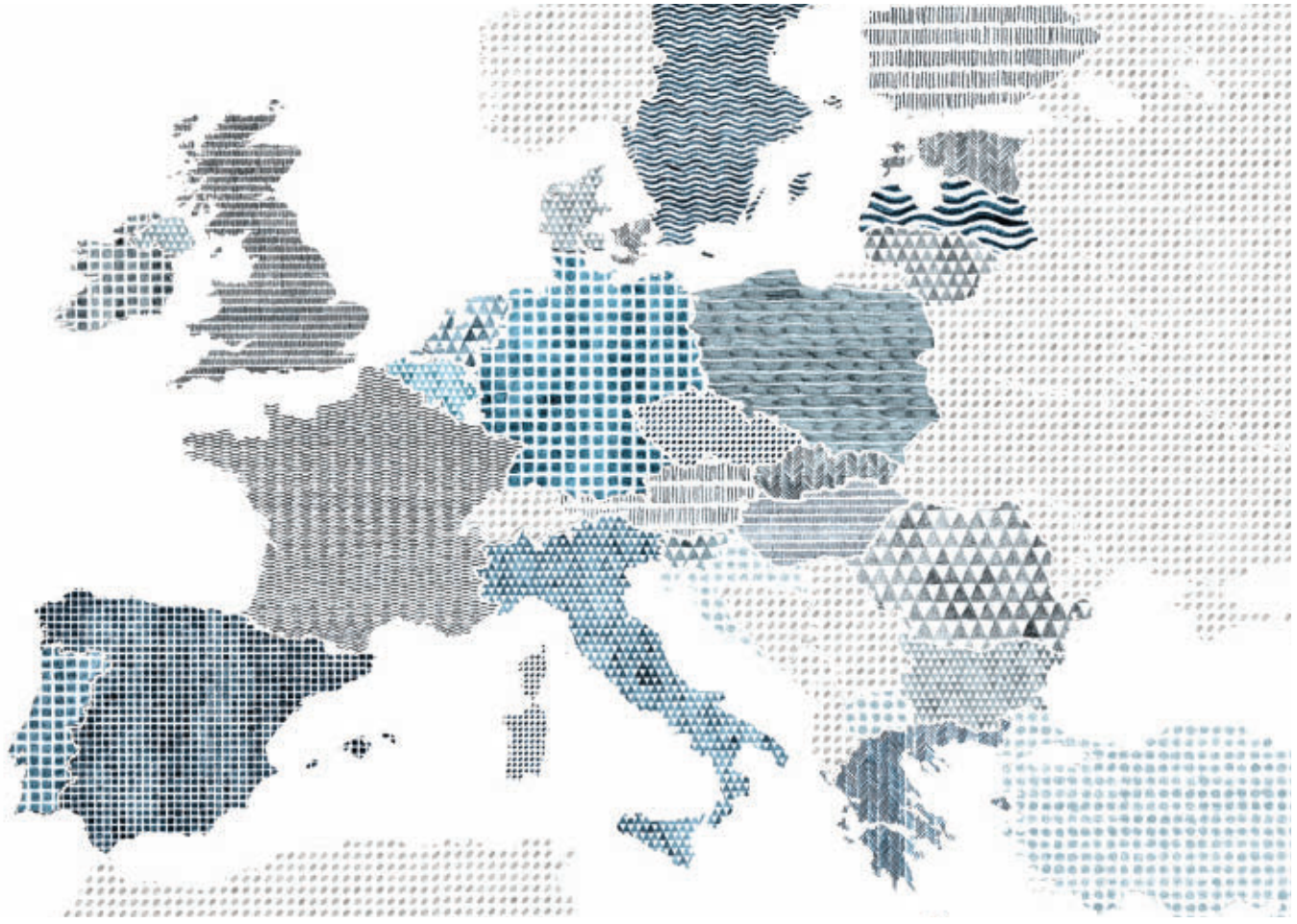
Differences in telecommunications regulation between the U.S. and the European Union are a key factor in viewing the network neutrality discussion from a European perspective.

Readers of this magazine will be familiar with the network neutrality debate currently occurring in the U.S. The February 2009 issue featured a Point/Counterpoint column by Barbara van Schewick and David Farber, respectively arguing in favor of and against legislative intervention to secure network neutrality (page 31). Many readers might have wondered whether the European Union has also been engulfed in the debate. The answer is yes, but as is often the case the EU and the U.S. are starting from different situations and working within different policy frameworks.

"Network neutrality" has become a slogan of sorts, which covers a more complex reality than either side of the U.S. debate is willing to admit. The key development that prompted the debate everywhere were statements by certain broadband Internet service providers that they wanted to move away from the "best-efforts" model currently prevailing. Instead of deploying best efforts to convey all the packets they handle to their destination (with delay, jitter, and so forth being distributed randomly), these ISPs would want to introduce differentiated quality of service (QoS) levels. Technically, ISPs would then need to inspect packets more intensively than they usually do in order to determine the QoS level with which to handle them.

In the EU as in the U.S., ISPs have two main reasons for desiring differentiated QoS. In the shorter term, it responds to perceived network management problems, in the wake of explosive Internet traffic growth with the rise of video-based applications, services, and content. For most ISPs today, a small fraction of their users (usually less than 10%) account for most of the use of their networks (usually around 80%). This imbalance is not reflected in the subscription rates, even though that small fraction of users generates network management problems that affect the quality of service provided to other users. Differentiated QoS—as a network management tool—would enable ISPs to correct some of that imbalance, since users (including application, service, and content providers) would then decide how much quality of service (priority) they want to purchase and their traffic would be treated accordingly. In economic terms, it is too early to tell whether such a development will increase welfare. In theory, tailoring QoS more closely to the preferences of each user is an improvement, but in practice the verdict will depend on the extent to which the users who opt for lower QoS offerings are properly compensated if—as is likely—they experience an inferior level of service.

The ISP landscape in Europe looks different than in the U.S. and is likely to remain so in the foreseeable future.



In the longer term, differentiated QoS can have much larger implications by affecting the balance of power between ISPs, their users, and content providers (including also service providers such as Google or application providers such as Skype). ISPs are under pressure to deliver ever faster connections to users and content providers, yet Internet access is becoming a commodity, with the price of subscriptions falling steadily. The trend can be reversed by turning the ISP network into a "platform," that is, offering specific QoS and performance levels to users and content providers alike, thereby making the ISP attractive to deal with ("the best video delivery," "the best gaming experience"), as opposed to just one of many alternatives on the market.

In addition, by positioning itself as a distinctive "platform," an ISP should be able to maintain, if not expand, its revenue stream; at a time when ISPs must carry out considerable investment in upgrading their networks (fixed and mobile alike), this could be a welcome evolution. Yet this would also imply a reshuffling of innovation patterns. So far the Internet has been very successfully driven through innovation "at the edge," outside of the networks (consider Google, Amazon, Skype, iTunes, and all the Web 2.0 providers). In the future, innovation could equally be coming from the ISPs on their platforms. It is not clear for now whether this will substitute for or complement innovation at the edge, that is, whether innovation at the edge will be reduced (because innovative upstarts would be shut out) or further spurred.

What is more, technically no one knows yet how such differentiated QoS offerings could be implemented across the various networks that typically make up the fabled Internet cloud. This brings me to mention some significant differences between the EU and the U.S. In the U.S., the provision of broadband Internet access is concentrated in a few hands, namely those of the remaining local exchange carriers providing ADSL (AT&T, Verizon, Qwest) and the large cable TV providers. The official FCC policy is to bank on competition between the relatively few providers of these infrastructure platforms (ADSL, cable, mobile). The Internet cloud would then give way to a limited number of single-firm platforms, each controlled by one of these providers, with two or more platforms being present at any given location in the U.S.

In the EU, the prospects for infrastructure competition are dimmer, since only a few areas (Benelux, parts of France, Germany, and the U.K.) are now served by competing broadband infrastructures (cable and ADSL). In most of the EU, it is thought that the rollout of competing broadband networks—effectively from scratch—cannot be achieved without some form of access to incumbent networks, at least in a starting phase. This means the ISP landscape in Europe looks different than in the U.S. and is likely to remain so in the foreseeable future: fewer competing infrastructures, but more market players, many of which rely on access to the incumbent's network. Furthermore, that landscape is structured along national lines. In the end, it is difficult to conceive how differentiated QoS could be successfully

introduced on single-firm platforms in the EU. More likely than not, significant coordination—through consortia/alliances, industrywide standardization or both—will be needed.

It is against that background that EU policymakers are considering whether to intervene. Their toolkit is different from that of their counterparts in the U.S. The EU regulatory framework for electronic communications (telecommunications) is formulated as a set of policy objectives, which national regulatory agencies implement with the help of instruments defined at the European level. Regulation must be based on sound economic analysis (as opposed to technology or history), and it is meant to be used only when it provides added value over the application of competition law. The regulatory framework is intended to be robust and sustainable without constant legislative intervention. In a sense, the discussion of network neutrality is a good test of these principles. So far, the dominant view is that the various issues raised by the introduction of differentiated QoS can largely be dealt with using existing legislation.

Indeed, many of the concrete difficulties experienced so far fall under EU competition law. For instance, in the U.S., the FCC inquired into the practices of Madison River—an ADSL provider that blocked access to voice over IP providers competing with its telephone service; and of Comcast—the large cable provider that blocked peer-to-peer traffic potentially competing with its cable TV service. In the EU, if an incumbent or any other ISP with enough market power to be found dominant engaged in a similar practice, it would most likely run afoul of Article 82 EC, which prohibits abuses of such dominant position (conduct that undermines competition by excluding competitors from the market).

Similarly, a dominant ISP would likely breach Article 82 EC if it attempted to create a walled garden or gated community whereby its own or affiliated content, applications, or services would be favored over those of competitors. If competition law were found not to have enough bite, then the regulatory regime specifically concerned with dominant operators (operators with significant market power or SMP) could be made applicable to the market for the transmission of content over the Internet. National regulatory agencies would then have the power to impose access and nondiscrimination obligations, in line with the EC regulatory framework.

The regulatory debate surrounding the introduction of differentiated QoS and network neutrality in Europe is not over by any means.

Furthermore, if all ISPs were to engage in blocking to such an extent that the Internet became "patchy" and its ability to deliver benefits to society was impaired, the current regulatory framework also offers a possibility to intervene to restore interconnectivity. Yet any intervention on this point would need to be very finely tuned: introducing differentiated QoS to improve network management implies that some users will choose not to purchase the top level of service, without them being in any way blocked from accessing what they desire.

In the end, even if the introduction of differentiated QoS entails some risks in addition to the benefits it could bring, it is too early to tell, and at this moment the case against differentiated QoS is not solid enough to warrant specific legislative intervention to impose network neutrality in the EU. The most important open issue for now is that subscribers know which QoS level they are getting from their ISP. Unfortunately, this is not always explained properly by ISPs.

As it turned out, the network neutrality debate hit Europe just as the EU lawmakers were conducting a general review of telecommunications regulation. The European Commission carried out the review and in 2007 submitted legislative proposals to the Council (made up of Member State governments) and the European Parliament for enactment. The Commission proposed to introduce a general principle that end users should be able to access and distribute any lawful content and use any lawful applications and/or services of their choice and to require ISPs to inform their users of any limitations imposed on that right. It also reserved for itself the right to develop minimum QoS requirements to be imposed on ISPs, if necessary.

In first reading, the European Parliament brought these proposals much further by framing the issue as a matter of fundamental rights and entrusting national regulatory agencies directly with the ability to introduce minimum QoS requirements. Yet the Member States, meeting in the Council, are much more prudent, and at the time this column was written, their view appears likely to prevail when the legislative process ends later in 2009. Contrary to the Commission and the Parliament, the Member States do not want at this time to enshrine any principle that users should have access to content, applications, and services of their choice. They would, however, require ISPs to inform users of traffic management policies and QoS levels. Finally, they would follow the Parliament in empowering national regulatory agencies to introduce minimum QoS requirements.

The regulatory debate surrounding the introduction of differentiated QoS and network neutrality in Europe is not over by any means. Legislative intervention for the time being is likely to be limited to strengthening transparency toward consumers, with the threat of minimal QoS requirements if the evolution took a turn for the worse. For the rest, the current regulatory framework will undoubtedly be used to deal with problems as they arise in specific cases. The next legislative review, probably in 2012, will then take stock of developments and lead to more definitive and informed legislative proposals if needed. C

Pierre Larouche (pierre.larouche@uvt.nl) is Professor of Competition Law and the director of the Tilburg Law and Economics Center (TILEC) at Tilburg University, The Netherlands.

DOI:10.1145/1506409.1506417
LeAnne Coder, Joshua L. Rosenbloom, Ronald A. Ash, and Brandon R. Dupont

Economic and Business Dimensions

Increasing Gender Diversity in the IT Work Force

Want to increase participation of women in IT work? Change the work.

It is commonly understood that the IT work force lacks gender diversity. In 1983 women made up approximately 43% of the IT work force according to the U.S. Bureau of Labor Statistics Current Population Survey. By 2008, while the total IT work force had more than doubled, the female percentage had dropped to 26%. In comparison, women represented approximately 46% of administrative, science, and technical workers and approximately 42% of all other occupations. A variety of explanations have been offered to account for the small share of women in IT. But based on our research[4, 5] we believe choice plays an important role in explaining why there are so few women in IT, and this in turn has important policy implications for what kinds of interventions will be effective in encouraging more women to enter IT.

Encouraging more women and minorities to choose IT careers would help raise the numbers in the field. Beyond this, however, increasing the diversity of IT will produce additional benefits by ensuring that IT professionals have a broad range of experience and interests. As Wulf has argued, "...those differences in experience are the "gene pool" from which creativity springs."[6]

The dearth of females in IT fields is part of a larger phenomenon of occupational segregation by gender. Explanations for these occupational differences can be grouped under three broad headings: discrimination; differences in ability; and choice. Identifying the reasons so few women enter IT careers is not simply an academic exercise; it also suggests some possible solutions that may help to rectify this situation.

In the past few years a number of pilot efforts have been undertaken to address a variety of perceived obstacles to women's participation in IT. These policy initiatives have focused on a variety

ILLUSTRATION BY JON HAN

of ways the problem of underrepresentation might be addressed. We think these policy proposals must, however, be informed by a clear understanding of the underlying reasons for the limited numbers of women in IT careers.

The KU Professional Worker Career Experience Study

To shed light on how men and women make career choices we conducted four in-depth focus groups with IT professionals in the greater Kansas City area, and then collected detailed information from a sample of over 500 IT and non-IT professionals. Participants in the survey were solicited from employees at several large organizations with offices in the central U.S. and from business school and computer science alumni of a large Midwestern university.

We sought to compare the family backgrounds, work histories, educational experiences, and personality characteristics of IT professionals with those of individuals working in equally demanding careers that required roughly comparable levels of education and skills. This quasi-experimental design allowed us to isolate the reasons for gender-based differences in career choice.

The sample consists of 523 working professionals. The non-IT professionals include accountants, auditors, CEOs, CFOs, presidents, consultants, engineers, managers, administrators, management analysts, scientists, technicians, nurses, and teachers. The IT professionals include application developers, programmers, software engineers, database administrators, systems analysts, Web administrators, and Web developers.

The dearth of females in IT fields is part of a larger phenomenon of occupational segregation by gender.

About three-quarters of the sample (73%) are non-IT professionals, with the remainder being IT professionals. The overall sample is almost evenly divided between men (54%) and women (46%), but consistent with broader national patterns the IT workers were mostly male (68%), while the non-IT professionals were nearly evenly divided between men and women. The average age of participants in our survey was 39 years and they averaged 17 years of formal education (92% held four-year college degrees).

Personality Matters for Career Choice

Vocational psychologists have developed a way of quantifying the personality differences between individuals and how those differences affect the choice of occupation. This line of research began in 1927 when E.K. Strong developed the Strong Vocational Interest Bank (SVIB; now the Strong Interest Inventory, SII). By the 1950s, Holland had augmented Strong's work by introducing six basic occupational interest categories that closely resembled the dimensions found in research on vocational interests using the SVIB.

In 1974, the theories developed by Holland and by Strong were combined to create the Strong Interest Inventory, which is used to measure six general occupational themes (GOT) for both people and jobs, and this approach remains one of the leading tools used by career counselors to match individuals to careers. These six vocational types (RIASEC) are:

- Realistic (R) refers to a person's preference for activities that entail the explicit, ordered, or systematic manipulation of objects, tools, and machines.
- Investigative (I) refers to a person's preference for activities that entail the systematic or creative investigation of physical, biological, and cultural phenomena.
- Artistic (A) refers to a person's preference for activities that are ambiguous, free, non-systematic and that entail the manipulation of materials to create art forms or products.
- Social (S) refers to a person's preference to lead others or for activities that entail the manipulation of others to inform, train, develop, cure, or enlighten.
- Enterprising (E) refers to a person's preference for activities that entail the manipulation of others to attain organization goals or economic gain.
- Conventional (C) refers to a person's preference for activities that entail the explicit, ordered, systematic manipulation of data.

Career fields are often chosen when a person finds a career that "matches" his or her personality. For example, accountants typically score very high on the Conventional GOT. Accounting jobs typically involve a systematic approach to credits and debits and financial statements. Similarly, computer programmers typically score highly on the Realistic GOT. Programming requires a focus on concrete problem solving to abstract reasoning.

We know from decades of work by vocational psychologists that the occupational themes measured by the SII are not distributed equally between men and women. Men, for example, score higher on Realistic and Investigative themes, while women score higher in Artistic, Social, Enterprising, and Conventional themes.[1,2]

Our analysis of the survey data we collected indicates that more than two-thirds of the gender difference between IT professions and our control group can be accounted for by differences in the distribution of GOT scores between men and women.[4] Based on these figures we estimate that in the absence of systematic gender differences in the distribution of GOT scores the IT work force today would be close to 40% female, rather than the actual figure of 26%.

IT workers in our study had higher scores on the Realistic and Investigative GOT. As discussed previously, fewer women have these types of occupational personalities, preferring occupations higher in the other four GOTs. Women do not view IT professions as artistic, social, enterprising, or conventional so they choose other occupations they feel will better match their personality.

Another recent study, by David Lubinski and Camilla Persson Benbow[3] supports our conclusions. Their work found that among a group of mathematically precocious youths who have been followed for up to 20 years women and men make quite different career choices. They note that mathematically

talented women are typically endowed with more highly developed verbal-linguistic skills than are men of similar mathematical ability and this versatility encourages different career choices.

Finding Ways to Increase Female Participation in IT

Finding that differences in occupational personality appear to explain much of the gender difference in career choice does not mean it is impossible to increase the number of women entering IT careers. Our discussions with focus group participants indicated there are important differences in how men and women entered IT, and that these offer a number of possible routes through which it may be possible to address current gender imbalances in IT.

Many of our focus group participants felt they had "fallen into" their IT careers, coming into IT by way of another career field. More systematic results from our survey echo this observation. Women in IT were significantly less likely than men or than women in non-IT careers to say their current career choice had been influenced by courses they had taken in high school or their high school teachers.

Focus group participants told us they discovered they had a natural aptitude for IT that led them to their current career field. Only six out of the 16 women in the focus groups actually had computer science degrees, suggesting the importance of maintaining multiple routes into IT professions.

In addition, conversations with the focus group participants emphasized that there are many misconceptions regarding what IT professionals actually do and that many IT jobs actually require occupational personalities that are more common among women. Several focus group participants mentioned they found the reality of their IT jobs to be different from what they had anticipated. These participants observed that their jobs often required them to act as a translator between the end user and the person actually writing the program code, something that made the job more social.

Their experiences suggest many IT jobs can be redesigned in ways that are more attractive to women by emphasizing the artistic, social, and conventional dimensions of the tasks they require.

There are many women in other professions with the requisite skills needed to succeed in IT.

There are many women in other professions with the requisite skills needed to succeed in IT. But recruiting them will require careful thought about how job responsibilities are structured and communicated. The benefits of this effort will be a more diverse and creative IT work force. C

References

1. Donnay, D.A.C., Morris, M.L., Schaubhut, N.A., and Thompson, R.C. *Strong Interest Inventory Manual, Revised Edition.* CPP, Inc., Mountain View, CA, 2004.
2. Holland, J.L. *Making Vocational Choices: A Theory of Vocational Personalities and Work Environments, Third Edition.* Lutz, Psychological Assessment Resources, 1997.
3. Lubinski, D. and Benbow, C.P. Study of mathematically precocious youth after 35 years: Uncovering antecedents for the development of math-science expertise. *Perspectives on Psychological Science 1*, 4 (Apr. 2006), 316–345.
4. Rosenbloom, J. L., Ash, R.A., Coder, L., and Dupont, B. Why are there so few women in information technology? Assessing the role of personality in career choices. *Journal of Economic Psychology 29*, 4 (Apr. 2008), 543–554.
5. Rosenbloom, J.L., Ash, R.A., Dupont, B.R., and Coder, L. Examining the obstacles to broadening participation in computing: Evidence from a survey of professional workers. *Contemporary Economic Policy.* (Forthcoming).
6. Wulf, W.A. Diversity in engineering. In *Moving Beyond Individual Programs to Systemic Change.* Women in Engineering Programs and Advocates Network Member Services, West Lafayette, IN, 1999.

LeAnne Coder (leanne.coder@wku.edu) is an assistant professor of Business at the University of Western Kentucky.

Joshua L. Rosenbloom (jrosenbloom@ku.edu) is Associate Vice Provost, Research and Graduate Studies and a professor of Economics at the University of Kansas and Research Associate, National Bureau of Economic Research.

Ronald A. Ash (rash@ku.edu) is a professor of Business at the University of Kansas.

Brandon R. Dupont (Brandon.Dupont@wwu.edu) is an assistant professor of Economics at Western Washington University.

This material is based upon work supported by the National Science Foundation under Grant No. 0204464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Calendar of Events

May 16–24
International Conference on Software Engineering,
Vancouver, Canada,
Sponsored: SIGSOFT,
Contact: Stephen Fickas,
Email: fickas@cs.uoregon.edu

May 18–20
Computing Frontiers Conference,
Ischia, Italy,
Sponsored: SIGMICRO,
Contact: Gerald R Johnson,
Email: gerry_johnson@yahoo.com

May 27–29
The Second International Conference on Immersive Telecommunications,
Berkeley, CA,
Contact: Ruzena R. Bajcsy,
Email: bajcsy@eecs.berkeley.edu

May 28–30
2009 Computer Personnel Research Conference,
Limerick, Ireland,
Contact: Norah Power,
Email: norah.power@ul.ie

May 31–June 2
Symposium on Theory of Computing Conference,
Bethesda, MD,
Contact: Aravind Srinivasan,
Email: srin@cs.umd.edu

June 1–4
CFP '09: Computers, Freedom and Privacy,
Washington Metro North Area DC,
Sponsored: PROFESSIONAL,
Contact: Cindy Southworth,
Email: cs@nnedv.org

June 3–5
Euro American Conference on Telematics and Information Systems,
Prague, Czech Republic,
Contact: Miroslav Svitek,
Email: svitek@fd.cvut.cz

June 3–5
The 19th International Workshop on Network and Operating Systems Support for Digital Audio and Video,
Williamsburg, VA,
Contact: Wei Tsang Ooi,
Email: ooiwt@comp.nus.edu.sg

DOI:10.1145/1506409.1506419

Martin Campbell-Kelly

Historical Reflections

The Rise, Fall, and Resurrection of Software as a Service

A look at the volatile history of remote computing and online software services.

One of the more hyped commercial opportunities these days appears to be software as a service or SaaS. In this form of computing, a customer runs software remotely, via the Internet, using the service provider's programs and computer infrastructure. One of the first and most successful firms in the SaaS space is Salesforce.com, which was launched in 1999. Salesforce.com provides a customer-relationship management service. Using the service, a mobile salesperson, for example, can access the software from a laptop while on the road, and the head office is relieved of all the problems of infrastructure provision, the complexities of managing and upgrading software, and synchronizing data from multiple sources. Another big player is Google, which now offers email and office productivity applications in its version of cloud computing.

Salesforce marketing campaign.

PHOTOGRAPH BY LINDA LANE

Many people think that the future of software lies in SaaS and cloud computing. They may well be right in the medium term, but history shows that one cannot be sure that the trend will last indefinitely.

There are two main components to SaaS: The software itself and the computing infrastructure on which it runs. Customers are at least as concerned about the quality of service as they are about the software. Indeed, for providers who use freely available open source software, quality of service is their only competitive advantage.

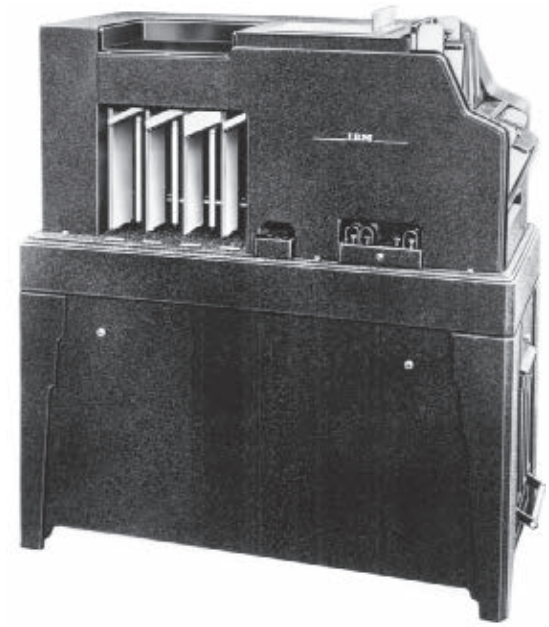
Organizations use in-house computing facilities or SaaS largely according to the economics of the situation—whether it is cheaper to own one's software and infrastructure or to buy services on-demand. This dilemma is not new. It is as old—indeed, older—than the computer industry itself.

Before computers came on the scene in the mid-1950s, the most advanced information processing equipment that organizations could buy (or lease) was punched-card electric accounting machines, or EAMs. The main vendor of this type of equipment, IBM, opened the first of several service bureaus in 1932. Customers brought their data processing needs to a bureau and came

Timesharing thrived just as long as its cost and convenience was competitive with a mainframe computer installation. The arrival of the PC changed everything.

back later for the results. The bureau provided customers with advanced information processing on-demand, thereby eliminating the cost of maintaining and staffing an EAM installation. Depending on the volume of data to be processed, using a service bureau tended to be more expensive per transaction than using one's own installation. Users had a choice. If one had a low volume of transactions then the economics favored the service bureau, but if one had a high volume of transactions it was cheaper to have one's own installation.

In 1949 a small firm, Automatic Payrolls Inc., was founded in New Jersey and used a variant of the service bureau business model. The firm specialized in payroll processing. It developed its own procedures—at first using bookkeeping machines, and then punched-card machines that were programmed with plug-boards. It would send a van to its customers to collect time sheets or punched cards, process the data, and drop off the results to its customers later. This made excellent business sense not only for organizations that did not want to maintain a bookkeeping machine or an EAM installation, but also for firms that simply wanted to offload the non-core activity of managing the payroll. In 1958, the company changed its name to Automatic Data Processing Inc., or simply ADP, and in 1961 it acquired an IBM 1401 computer. ADP expanded into new locations and by the mid-1960s it was using the emerging capabilities of data communications to eliminate some of the physical collection and return of data.



Introduced in 1937, the IBM 77 collator rented for $80 a month. It was capable of handling 240 cards a minute, and was 40.5 inches long and 51 inches high.

Top: Henry Taub (left) in ADP's first computer room. Bottom: Teletype.

FROM TOP PHOTOGRAPH COURTESY OF IBM. ADP. DAVID L. MILLS

Many other firms began to compete with ADP, offering different services in what became the biggest sector of the "data processing services industry." In 1961 the industry formed its own trade association, ADAPSO—the Association of Data Processing Services Organizations, the ancestor of today's ITAA. By 1970 processing services accounted for more than one-quarter of total U.S. computing purchases. While firms have come and gone, ADP seems to have found the perfect niche—today it is still the world's biggest payroll processor, preparing the paychecks for one-sixth of the total U.S. work force.[a]

In the mid-1960s timesharing computers came on the scene. In these systems customers could access a mainframe computer remotely. Connected to a mainframe computer via a regular telephone line, users ran programs using a clunky, 10-characters-per-second, model ASR-33 teletype. It made for a noisy working environment, but on-demand computing had real benefits. Salespeople for the timesharing firms touted their systems using the computer-utility argument: Firms did not maintain their own electric plants, it was argued, instead, they bought power on-demand from an electric utility; likewise, firms should not maintain mainframe computers, but instead get computing power from a "computer utility." Several national computer utility companies had emerged by the end of the 1960s. But then came the first computer recession in 1970. The computer utility model turned out to be very vulnerable to an economic downturn. Similar to the way firms cut back on discretionary travel during a recession, they also reduced spending on computer services. There were many firm failures and bankruptcies. For example, one of the most prominent firms, University Computing—which had computer centers in 30 states and a dozen countries—saw its revenues hemorrhage, and its stock price dramatically declined from a peak of $186 to $17.

The timesharing industry recovered, however. In the 1970s major players included General Electric, Timeshare Inc., and CDC. They built massive global computer centers that serviced thousands of users. By then those clunky teletypes had been replaced with visual display units, or "glass teletypes" as they were sometimes known. They were silent and relatively pleasant to use, giving an experience somewhat like using an early personal computer. Increasingly firms sought to differentiate their offerings by providing exclusive software. For example, they devised financial analysis programs that can now be seen as forerunners of spreadsheet software. They implemented some of the first email systems. They also hosted the products of the independent software industry, usually paying them on a royalty basis, with typically 20% of revenues going to the software provider.

The timesharing industry died a second time around 1983–1984. This time it was not a computer recession that was the cause, but the personal computer.

The timesharing industry died a second time around 1983–1984. This time it was not a computer recession that was the cause, but the personal computer. Timesharing services cost $10 to $20 per hour, with regular users billing perhaps $300 a month. The PC completely destroyed the economic basis of the timesharing industry. Compared with a timesharing service, a PC would pay for itself in well under a year, and it had the further advantages of eliminating the telephone connection and providing an instantaneous response. Furthermore, a standalone PC was not like a mainframe computer—it was a fuss-free, virtually maintenance-free, piece of office equipment. As the timesharing industry went into decline, a few of the firms morphed into consumer networks, such as CompuServe and GE's Genie, but mostly they just faded away with their vanishing revenues.[b]

Today, the very things that killed the timesharing industry in the 1980s have been reversed. Despite falling hardware costs, computing infrastructure has become increasingly complex and expensive to maintain—for example, having to deal with security issues and frequent software upgrades. Conversely, communications costs have all but disappeared compared with the 1980s. No wonder remote computing is back on the agenda.

Cloud computing has many parallels with the 20-year reign of timesharing systems. Timesharing thrived just as long as its cost and convenience was competitive with a mainframe computer installation. The arrival of the PC changed everything. Today, cloud computing offers tremendous advantages over the in-house alternative of maintaining a cluster of servers, application programs, and database software. However, if the cost of maintaining this infrastructure was to fall dramatically (which is entirely possible in the next few years) the economic advantage of cloud computing could be reversed. The other threat to cloud computing is a major economic downturn. Now that U.S. industry experiencing a recession, the demand for remote computing could decline, just like the demand for electric power. Further, many online services are currently funded by advertising revenues—take away the demand for advertising and there will be little to support these services.

Of course, none of the aforementioned items should be construed as a forecast of the impending demise of software as a service. Rather, this column is intended as a salutary reminder that nothing in IT lasts forever, and that technological evolution and economic factors can rapidly alter the trajectory of the industry. C

a For a history of ADP Inc. see: *ADP Fiftieth Anniversary 1949–1999*; http://www.investquest.com/iq/a/adp/main/archives/anniversary.htm#.

b For a history of the timesharing industry see: M. Campbell-Kelly and D.D. Garcia-Swartz, "Economic Perspectives on the History of the Computer Timesharing Industry, 1965–1985," *IEEE Annals of the History of Computing 30*, 1 (Jan. 2008), 16–36.

Martin Campbell-Kelly (M.Campbell-Kelly@warwick.ac.uk) is a professor in the Department of Computer Science at the University of Warwick, where he specializes in the history of computing.

DOI:10.1145/1506409.1506420

Mark Guzdial

Education
Teaching Computing to Everyone

Studying the lessons learned from creating high-demand computer science courses for non-computing majors.

Several computing programs in the U.S. are developing new kinds of introductory computing courses for non-computing majors, some with support from the NSF CPATH program. At Georgia Institute of Technology (Georgia Tech), we are entering our 10th year of teaching computing to every undergraduate on campus. Our experience gained during the last decade may be useful to others working to understand how to satisfy the growing interest in computing education across the academy.

Computing in General Education

In fall 1999, the faculty at Georgia Tech adopted a requirement that all students must take a course in computing. We modified the academic year from quarters to semesters, which gave the campus the opportunity to rethink the curriculum and our general education requirements. Russ Shackelford, Rich Leblanc, Kurt Eiselt, and the College of Computing's then-dean, Peter Freeman, convinced the rest of Georgia Tech that all students who graduated from an Institute of Technology should know computing. We started before publication of the National Research Council report *Being Fluent with Information Technology*,[3] though that report significantly influenced implementation.

The new requirement wasn't a hard sell. Faculty in the College of Engineering had wanted to implement a programming requirement for their students, but couldn't decide who should teach it. The creation of the College of Computing in 1990 answered the question of whose job it was to teach computer science at Georgia Tech. Faculty in the Ivan Allen College of Liberal Arts (and in other colleges) embraced the new requirement. Computing was increasingly relevant for their disciplines, and was a value-added requirement for their graduates. The campus administration was kept abreast and involved throughout to maintain support. The new general education requirement was defined as an outcome—students would be able "to make algorithmic and data structures choices" when writing programs. That simple phrase describes a serious introductory course.

Teaching Everyone in One Class

For the first four years of the requirement, only a single class met the requirement: CS1321. There were several reasons for having only a single course. While we were already teaching approximately two-thirds of the students at Georgia Tech (because several of the largest degree programs already required computing), teaching everyone on campus meant well over 1,200 students a semester. The immensity of the task was daunting—splitting our resources over several courses seemed a bad start-up strategy. We were also explicitly concerned about creating

PHOTOGRAPH BY JUDSON COLLIER

The Christopher W. Klaus Advanced Computing Building on the Georgia Tech campus is home to the Institute's College of Computing and School of Electrical and Computer Engineering.

a course that no faculty cared about. Courses just offered as a "service" get less attention. By putting all students in one class, it is in everyone's interest to ensure the class is good.

The class received significant faculty interest and used innovative curricula. We started out using Shackelford's pseudocode approach to learning.[7] Faculty in the other majors complained about students not gaining experience debugging programs. We later moved to Felleisen et al.'s *How to Design Programs* text using Scheme.[4] These were, and are, approaches for teaching computing that have been successfully used at many institutions.

By 2002, however, CS1321 may have been the most hated course on campus. From 1999 to 2002, the overall success rate (leaving the course with an A, B, or C—not counting those students who received a D, a failing grade, or withdrew from the course) was 78%. That's not too bad for an introductory computing course.[1] However, this was a course with everyone in it. When we examine those majors where a computing requirement is atypical, we see 46.7% of architecture students succeeding each semester, 48.5% in management, and 47.9% in public policy. We failed more than half of the students in those majors each semester; females failed at nearly twice the rate of males. Statistics like these are a concern for both the Georgia Tech and the College of Computing—it hinders our relations with the rest of campus when computing is the gatekeeper holding back their students.

Developing Contextualized Computing Education

Around this time, several studies were published critiquing computing courses, including the AAUW's *Tech-Savvy* report[2] and *Unlocking the Clubhouse* by Margolis and Fisher.[6] These reports describe students' experiences in computing as "tedious," "asocial," and surprisingly, "irrelevant." A 2002 task force, chaired by Jim Foley, found similar issues at Georgia Tech. How could computing be "irrelevant" when it pervades so much of our world? Perhaps the problem was that our course had little connection to the computing in our students' world. While students are amazed at the Web, handheld video games, and smartphones, most introductory courses introduce students to the computing concepts behind these wonders with Fibonacci numbers and the Towers of Hanoi. What students saw as computing was disconnected from what we showed them in our computing class.

We chose to teach computing in terms of practical domains (a "context") that students recognize as important.

We adopted an approach that we call contextualized computing education. We chose to teach computing in terms of practical domains (a "context") that students recognize as important. The context permeates the course, from examples in lecture, to homework assignments, and even to the textbooks specially written for the courses. We decided to teach multiple courses, to match majors to relevant contexts.

In spring 2003, the College of Computing began offering three different introductory computing courses. The first was a continuation of CS1321, aimed at computing and sciences majors. The second was a new course for students in the College of Engineering, with much the same content, but in MATLAB and using an engineering context. The third was a new course for students in the colleges of liberal arts, architecture, and management using a context of manipulating digital media.

The engineering course was developed jointly with faculty from the schools of aerospace, civil, mechanical, and chemical engineering. Several faculty members in these schools had already started developing an alternative to CS1321, using MATLAB, a common programming language in engineering. Their model involved small classes in a closed lab working on real engineering problems. That course was prohibitively expensive to ramp up to over 1,000 engineering students each semester. The engineering faculty worked with David Smith of the College of Computing to create a course that used their examples and MATLAB, but taught the same computing concepts as CS1321.[9]

The course around "media computation" was built with an advisory board of faculty from the colleges of liberal arts, architecture, and management. The board's awareness and support for the course was important in getting the course approved as fulfilling the computing requirement in programs of those colleges. The advisory board favored a programming language that was perceived as being easy to learn but was not associated with "serious" computer science. We chose the Python implementation, over concerns about both Scheme and Java.

Media computation is the context of how digital media tools like Photoshop and GIMP work. We created cross-platform libraries to manipulate pixels in a picture and samples in a sound. We taught, for example, iterating across an array by generating grayscale and negative versions of an image and array concatenation by splicing sounds. We were able to cover all the introductory computing concepts using media examples. In their homework, students created pictures, sounds, HTML pages, and animations. We created an integrated development environment that provided the media functions as well as tools for inspecting pictures and sounds.[5]

Impact of Contextualized Computing Education

Faculty and students are happier with the new courses. The success rates rose above 80% in both the engineering and media courses. When comparing success rates to those same majors mentioned previously, we found the average success rate in the first two years for architecture students rose to 85.7%, management to 87.8%, and public policy to 85.4% per semester. The media computation course has been majority female, and women succeed at the same or better rates than the male students. Similar improvements in success rates in media computation courses have been seen among underrepresented groups at other campuses.[8]

New opportunities appear on campus when all students succeed at com-

The Georgia Tech LWC Productivity Computer Cluster.

puting. We have introduced a minor in computer science. We had enough students interested in computing after the media course that we now offer a second course, on data structures within a media context. A second course was also developed for engineering students, so we now teach three second computing courses, as well as three introductory courses.

Faculty in the School of Interactive Computing and the School of Literature, Culture, and Communication (in the College of Liberal Arts) now offer a new joint undergraduate degree, a bachelor of science degree in computational media. The course was developed because of growing common interest in areas like video games, augmented reality, and computer animations. While the common research interests were clearly the motivating factor in deciding to create the new degree program, having a media computation course that could draw students into the new program from liberal arts, as well as from computing, facilitated the joint effort.

Developing contextualized courses is challenging and expensive, but the results can be shared.

We see an increasing number of courses around campus that require students to write programs, though not necessarily as an outcome of the computing requirement. Computing is growing in importance in all fields. Non-computing faculty request us to include particular concepts or tools in the introductory courses and to provide prerequisite knowledge and skills for advanced courses. In this way, the computing requirement has become part of curricula across campus.

In the first years, the success rates for the new courses were sometimes higher than the success rate in the continuing CS1321. We realized that even computer science majors need introductory courses that connect explicitly to a context that students recognize as computing. In a joint effort with Bryn Mawr College and with funding by Microsoft Research, we launched the Institute for Personal Robotics in Education (IPRE, http://www.roboteducation.org) to develop a new introductory course that uses robotics as the context for teaching introductory computing.

Lessons Learned

We in the College of Computing believe the use of contextualized computing education has been a significant step in making Georgia Tech's universal computing requirement successful. Developing contextualized courses is challenging and expensive (for example, writing textbooks, developing new integrated development environments), but the results can be shared. Other campuses are adopting our contextualized approaches, and some are developing their own.

We recommend involving faculty from the other departments in building courses for non-major students. They understand their students' needs in later courses and in their students' future professions. Further, we need them as context informants as we develop courses that teach through examples from their domains.

Finally, building successful, high-demand courses for non-computing majors gives us a different perspective on the current enrollment crisis. Students want these courses. Other schools on campus want to collaborate with us to build even more contextualized classes. While we still want more majors, we have an immediate need for more faculty time to develop and teach these courses that bring real computing to all students on campus. C

References

1. Bennedsen, J. and Caspersen, M.E. Failure rates in introductory programming. *ACM SIGCSE Bulletin 39*, 2 (2007), 32–36.
2. Commission on Technology, Gender, and Teacher Education. *Tech Savvy: Educating Girls in the New Computer Age*, American Association of University Women, 2000.
3. Committee on Information Technology Literacy, National Research Council. *Being Fluent with Information Technology*. The National Academies Press, 1999.
4. Felleisen, M., Findler, R.B., Flatt, M., and Krishnamurthi, S. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
5. Guzdial, M. *Introduction to Computing and Programming in Python, A Multimedia Approach*. Prentice Hall, 2005.
6. Margolis, J. and Fisher, A. *Unlocking the Clubhouse: Women in Computing*. MIT Press, 2001.
7. Shackelford, R.L. *Introduction to Computing and Algorithms*. Addison Wesley, 1997.
8. Sloan, R.H. and Troy, P. CS 0.5: A better approach to introductory computer science for majors. *ACM SIGCSE Bulletin 40*, 1 (2008), 271–275.
9. Smith, D.M. *Engineering Computation with MATLAB*. Addison Wesley, 2007.

Mark Guzdial (guzdial@cc.gatech.edu) is a professor in the College of Computing at Georgia Institute of Technology in Atlanta, GA.

PHOTOGRAPH BY CINDI TRAINOR

DOI:10.1145/1506409.1506421

Ken Birman and Fred B. Schneider

Viewpoint
Program Committee Overload in Systems

Conference program committees must adapt their review and selection process dynamics in response to evolving research cultural changes and challenges.

Major conferences in the systems community—and increasingly in other areas of computer science—are overwhelmed by submissions. This could be a good sign, indicative of a large community of researchers exploring a rich space of exciting problems. We're concerned that it is instead symptomatic of a dramatic shift in the behavior of researchers in the systems community, and this behavior will stunt the impact of our work and retard evolution of the scientific enterprise. This Viewpoint explains the reasoning behind our concern, discusses the trends, and sketches possible responses. However, some problems defy simple solutions, and we suspect this is one of them. So our primary goal is to initiate an informed debate and a community response.

The Growing Crisis

The organizers of SOSP, OSDI, NSDI, SIGCOMM,[a] and other high-ranked systems conferences are struggling to review rapidly growing numbers of submissions. Program committee (PC) members are overwhelmed. Good papers are being rejected on the basis of low-quality reviews. And arguably it is the more innovative papers that suffer, because they are time consuming to read and understand, so they are the most likely to be either completely misunderstood or underappreciated by an increasingly error-prone process. These symptoms aren't unique to systems, but our focus here is on the systems area because culture, traditions, and values differ across fields even within computer science—we are wary of speculating about research communities with which we are unfamiliar.

The sheer volume of submissions to top systems conferences is in some ways a consequence of success: as the number of researchers increases, so does the amount of research getting done. To have impact—on the field or the author's career—this work needs to be published. Yet the number of high-quality conferences cannot continue

a ACM Symposium on Operating Systems Principles (SOSP), ACM-USENIX Symposium on Operating Systems, Design and Implementation (OSDI), ACM Symposium on Networked Systems Design and Implementation (NSDI); the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM). This is a partial list and includes at most half of the high-prestige conferences in our field.

growing in proportion to the number of submissions and still promise presenters an influential audience, because there are limits on the number of conferences that researchers can attend. So attention by an ever-growing community necessarily remains focused on a small set of conferences.

The high volume of submissions is also triggering a second scaling problem: the shrinking pool of qualified and willing PC candidates. The same trends that are making the field exciting also bring all manner of opportunities to top researchers (who are highly sought as PC members). Those who do serve on PCs rightly complain that they are overworked and unable to read all the submissions.

- If submissions are read by only a few PC members then there will be fewer broad discussions at PC meetings about the most exciting new research directions. Yet senior PC members often cite such dialogue as their main incentive for service.
- If fewer senior researchers are present at the PC meeting then serving on the PC no longer provides informal opportunities for younger PC members to interact with senior ones.

And a growing sense that the process is broken has begun to reduce

We see a confluence of factors that amplify—increasing the magnitude without adding content to a signal—the pool of submissions.

the prestige associated with serving on a PC. Service becomes more of a burden and less likely to help in career advancement. When serving on a PC becomes unattractive, a sort of death spiral is created.

In the past, journal publications were mandatory for promotions at the leading departments. Today, promotions can be justified with publications in top conferences (see, for example, the CRA guidelines on tenure[b]). Yet conference publications are shorter. This leads to more publications per researcher and per project, even though the aggregate scientific content of all these papers is likely the same (albeit with repetition for context-setting). So our current culture creates more units to review with a lower density of new ideas.

Conference publications are an excellent way to alert the community to a general line of inquiry or to publicize an exciting recent result. Nevertheless, we believe that journal papers remain the better way to document significant pieces of systems research. For one thing, journals do not force the work to be fractured into 12-page units. For another, the review process, while potentially time consuming, often leads to better science and a more useful publication. Perhaps it is time for the pendulum to swing back a bit.

Looking Back and Peering Ahead

How did we get to this point? Historically, journals accepted longer papers and imposed a process involving multiple rounds of revision based on careful review. Publication decisions were made by standing boards of editors, who are independent and reflective. So journal papers were justifiably perceived as archival, definitive publications. And thus they were required for tenure and promotions.

This pattern shifted at least two decades ago, when the systems researchers themselves voted with their feet. Given the choice between writing a definitive journal paper about their last system (having already published a paper in a strong conference) versus building the next exciting system, systems researchers usually opted to build that next system. Computer science departments couldn't face having their promising young leaders denied promotion over a lack of journal publications, so they educated their administrations about the unique culture of the systems area. With journal publication no longer central to career advancement, increasing numbers of researchers chose the path offering quicker turnaround, less dialogue with reviewers, and that accepted smaller contributions (which are easier to devise and document).

As submissions declined, journals started to fill their pages by publishing material from top conferences. Simultaneously, under cost pressure, journals limited paper lengths, undercutting one of their advantages. Reviewers for journals receive little visibility or thanks for their efforts, so it is a task that often receives lower priority. And that leads to publication delays that some researchers argue make journal publication unattractive, although when ACM TOCS[c] (a top systems journal) reduced reviewer delay, researchers remained resistant to submitting papers there.

Simultaneously, the top conferences have also evolved. Once, SOSP and SIGCOMM were self-policed: submissions were not blinded, so submitting immature work to be read by a program committee populated by the field's top researchers could tarnish your reputation. And the program committees read *all* the submissions, debating each acceptance decision (and many rejections) as a group. An

b See http://www.cra.org/reports/tenure_review.html.

c *ACM Transactions on Computer Systems* (TOCS).

author learned little about that debate, though, receiving only a few sentences of hastily written feedback with an acceptance or rejection decision.

Today, author names are hidden from the program committee, the top conferences provide authors of all submissions detailed reviews, and there are more top conferences (for example, OSDI and NSDI) for an author to target. So authors feel emboldened to submit almost any paper to almost any conference, because acceptance will advance their research and career goals, but rejection does them virtually no harm. In fact, a new dynamic has evolved, where work is routinely submitted in rough, preliminary form under a mentality that favors a cycle of incremental improvements based on the detailed program committee feedback until the work exceeds the acceptance threshold of some PC. And often that threshold is reached before the work is fully refined. Thus, it is not uncommon to see publication of an initial paper containing a clever but poorly executed idea, a much improved follow-on paper published elsewhere, and then a series of incremental results being published. Perversely, this maximizes author visibility but harms the broader scientific enterprise.

Thus we see a confluence of factors that amplify—increasing the magnitude without adding content to a signal—the pool of submissions. Faced with huge numbers of papers, it is inevitable that the PC would grow larger, that reviewing would be done outside the core PC, or that each PC member would write reviews for only a few papers. The trend toward Web-based PCs that don't actually meet begins to look sensible, because it enables ever-larger sets of reviewers to be employed without having to assemble for an actual meeting. Indeed, even in the face-to-face PC model, it is not uncommon for the PC meeting to devolve into a series of subgroup discussions, with paper after paper debated by just two or three participants while 20 others read their email.

A solution must accommodate a field that is becoming more interdisciplinary in some areas and more specialized in others.

Reviews written by non-PC members, perhaps even Ph.D. students new to the field, introduce a new set of problems. What does it mean when an external reviewer checks "clear accept" if he or she has read just two or three out of 200 submissions and knows little of the prior work? The quality rating of a paper is often submerged in a sea of random numbers. Yet lacking any alternative, PCs continue to use these numbers for ranking paper quality. Moreover, because authorship by a visible researcher is difficult to hide in a blinded submission (and such an author is better off not being anonymous), work by famous authors is less likely to experience this phenomenon, amplifying a perception of PC unfairness.

Faced with the painful reality of large numbers of submissions to evaluate, PC members focus on flaws in an effort to expeditiously narrow the field of papers under consideration. Genuinely innovative papers that have issues, but could have been conditionally accepted, are all too often rejected in this climate of negativism. So the less ambitious, but well-executed work trumps what could have been the more exciting result.

Looking to the future, one might expect electronic publishing in its many manifestations to reshape conference proceedings and journal publications, with both positive and negative consequences. For example, longer papers can be easily accommodated in electronic forums, but authors who take advantage of this option may make less effort to communicate their findings efficiently. The author submits camera-ready material, reducing production delays, but the considerable value added by having a professional production and editing staff is simultaneously lost.

As the nature of research publication evolves, the community needs to con-

template two fundamental questions:

▸ What should be the nature of the review and revision process? How rigorous need it be for a given kind of publication venue? Should a dialogue involving referees' reviews and authors' revisions plus rebuttals be required for all publication venues or just journals? How should promotion committees treat publication venues—like conferences—where acceptance is highly competitive but the decision process is less deliberative and nobody scrutinizes final versions of papers to confirm that issues were satisfactorily resolved? How do we grow a science where the definitive publications for important research are neither detailed nor carefully checked?

▸ Should we continue to have high-quality, "must-attend" conferences, with the excitement, simultaneity, and ad hoc in-the-halls discussions that these bring? If we do, and they remain few in number, does it make sense for these to be structured as a series of plenary sessions in which (only) the very best work is presented? As an alternative, conferences could make much greater use of large poster sessions or "brief presentation" sessions, structured so that no credible submission is excluded (printing associated full papers in the proceedings). By offering authors an early path to visibility, could these kinds of steps reduce pressure?

A High-Level View: What Must Change (and What Must Not)

An important role—if not *the* role—of conferences and journals is to communicate research results: impact is the real metric. And in this we see some reason for hope, because a community seeking to maximize its impact would surely not pursue a strategy of publishing modest innovations rather than revolutionary ones. Force fields are needed to encourage researchers to maximize their impact, but creating these force fields will likely require changing our culture and values.

Another Viewpoint column[d] in this magazine suggested a game-based formulation of the situation, where the winning strategy is one that incentivizes both authors and program committees to behave in ways that remedy the problems discussed here. One can easily conjure other characterizations of the situation and other means of redress. But any solution must be broad and flexible, since systems research is far from a static enterprise. A solution must accommodate a field that is becoming more interdisciplinary in some areas and more specialized in others, challenging the very definition of "systems." For example, the systems research community is starting to embrace studying corporate infrastructure components that (realistically) can only be investigated in highly exclusive proprietary settings—publication and validation of results now brings new challenges.

Absent such steps or others that a communitywide discussion might yield, we shall find ourselves standing on the toes of our predecessors rather than on their shoulders.

Nevertheless, some initial steps to solving the field's problems are evident. Why not make a deliberate effort to evaluate accomplishments in terms of impact? To the extent that we are a field of professionals who advance in our careers (or stall) on the basis of rigorous peer reviews, such a shift could have a dramatic effect. We need to learn to filter CVs inflated by the phenomena discussed previously, and we need to publicize and apply appropriate standards in promotions, awards, and in who we perceive as our leaders.

Program committees need to adapt their behavior. Today, PCs are not only decision-making bodies for paper acceptances but they have turned into rapid-response reviewing services for any and all. If authors of the bottom two-thirds of the submissions did not receive detailed reviews, then there would be less incentive for them to submit premature work. And even if they did submit poorly developed papers, the workload of the PC would be substantially decreased given the reduced reviewing load. If some sort of reviewing service is needed by the field (beyond asking one's research peers for their feedback on a draft) rather than overloading our PCs, we should endeavor to create one—the Web, social networks, and ad hoc cooperative enterprises like Wikipedia surely can be adapted to facilitate such a service.

Finally, authors must revisit what they submit and where they submit it, being mindful of their obligation as scientists to help create an archival literature for the field. Early, unpolished work should be submitted to workshops or conference tracks specifically designed for cutting-edge but less validated results. Presentation of work at such a workshop should not preclude later submitting a refined paper to a conference. And publishing papers at a conference should not block submitting a definitive work on that topic for careful review and ultimate publication in an archival journal.

Absent such steps or others that a communitywide discussion might yield, we shall find ourselves standing on the toes of our predecessors rather than on their shoulders. And we shall become less effective at solving the important problems that lie ahead, as systems become critical in society. Older and larger fields, such as medicine and physics, long ago confronted and resolved similar challenges. We are a much younger discipline, and we can overcome those problems too. C

d J. Crowcroft, S. Keshav, and N. McKeown. Scaling the academic publication process to Internet scale. *Commun. ACM 52*, 1 (Jan. 2009), 27–30.

Ken Birman (ken@cs.cornell.edu) is the N. Rama Rao Professor of Computer Science in the Department of Computer Science at Cornell University, Ithaca, NY.

Fred B. Schneider (fbs@cs.cornell.edu) is the Samuel B. Eckert Professor of Computer Science in the Department of Computer Science at Cornell University, Ithaca, NY.

We are grateful to three *Communications* reviewers for their comments on our original submission; Jon Crowcroft, Robbert van Renesse, and Gün Sirer also provided extremely helpful feedback on an early draft. We are also grateful to the organizers and attendees of the 2008 NSDI Workshop on Organizing Workshops, Conferences and Symposia (WOWCS), at which many of the topics discussed in this Viewpoint were raised.

ACM, Uniting the World's Computing Professionals, Researchers, Educators, and Students

Dear Colleague,

At a time when computing is at the center of the growing demand for technology jobs worldwide, ACM is continuing its work on initiatives to help computing professionals stay competitive in the global community. ACM's increasing involvement in activities aimed at ensuring the health of the computing discipline and profession serve to help ACM reach its full potential as a global and diverse society which continues to serve new and unique opportunities for its members.

As part of ACM's overall mission to advance computing as a science and a profession, our invaluable member benefits are designed to help you achieve success by providing you with the resources you need to advance your career and stay at the forefront of the latest technologies.

I would also like to take this opportunity to mention ACM-W, the membership group within ACM. ACM-W's purpose is to elevate the issue of gender diversity within the association and the broader computing community. You can join the ACM-W email distribution list at http://women.acm.org/joinlist.

ACM MEMBER BENEFITS:

- A subscription to ACM's newly redesigned monthly magazine, ***Communications of the ACM***
- Access to ACM's **Career & Job Center** offering a host of exclusive career-enhancing benefits
- **Free e-mentoring services** provided by MentorNet®
- **Full access to over 2,500 online courses** in multiple languages, and 1,000 virtual labs
- **Full access to 600 online books** from Safari® Books Online, featuring leading publishers, including O'Reilly (Professional Members only)
- **Full access to 500 online books** from Books24x7®
- Full access to the new ***acmqueue*** website featuring blogs, online discussions and debates, plus multimedia content
- The option to subscribe to the complete **ACM Digital Library**
- The **Guide to Computing Literature**, with over one million searchable bibliographic citations
- The option to connect with the **best thinkers in computing** by joining **34 Special Interest Groups** or **hundreds of local chapters**
- **ACM's 40+ journals and magazines** at special member-only rates
- ***TechNews***, ACM's tri-weekly email digest delivering stories on the latest IT news
- ***CareerNews***, ACM's bi-monthly email digest providing career-related topics
- ***MemberNet***, ACM's e-newsletter, covering ACM people and activities
- **Email forwarding service & filtering service**, providing members with a free acm.org email address and **Postini** spam filtering
- And much, much more

ACM's worldwide network of over 92,000 members range from students to seasoned professionals and includes many of the leaders in the field. ACM members get access to this network and the advantages that come from their expertise to keep you at the forefront of the technology world.

Please take a moment to consider the value of an ACM membership for your career and your future in the dynamic computing profession.

Sincerely,

Wendy Hall

President
Association for Computing Machinery

Association for Computing Machinery

Advancing Computing as a Science & Profession

Association for Computing Machinery

Advancing Computing as a Science & Profession

membership application & *digital library* order form

Priority Code: ACACM10

You can join ACM in several easy ways:

Online	Phone	Fax
http://www.acm.org/join	*+1-800-342-6626 (US & Canada)* *+1-212-626-0500 (Global)*	*+1-212-944-1318*

Or, complete this application and return with payment via postal mail

Special rates for residents of developing countries:
http://www.acm.org/membership/L2-3/

Special rates for members of sister societies:
http://www.acm.org/membership/dues.html

Please print clearly

Name

Address

City | State/Province | Postal code/Zip

Country | E-mail address

Area code & Daytime phone | Fax | Member number, if applicable

Purposes of ACM

ACM is dedicated to:
1) advancing the art, science, engineering, and application of information technology
2) fostering the open interchange of information to serve both professionals and the public
3) promoting the highest professional and ethics standards

I agree with the Purposes of ACM:

Signature

ACM Code of Ethics:
http://www.acm.org/serving/ethics.html

choose one membership option:

PROFESSIONAL MEMBERSHIP:

- ❑ **ACM Professional Membership: $99 USD**
- ❑ **ACM Professional Membership plus the ACM Digital Library: $198 USD ($99 dues + $99 DL)**
- ❑ **ACM Digital Library: $99 USD (must be an ACM member)**

STUDENT MEMBERSHIP:

- ❑ **ACM Student Membership: $19 USD**
- ❑ **ACM Student Membership plus the ACM Digital Library: $42 USD**
- ❑ **ACM Student Membership PLUS Print *CACM* Magazine: $42 USD**
- ❑ **ACM Student Membership w/Digital Library PLUS Print *CACM* Magazine: $62 USD**

All new ACM members will receive an ACM membership card.
For more information, please visit us at www.acm.org

Professional membership dues include $40 toward a subscription to *Communications of the ACM*. Member dues, subscriptions, and optional contributions are tax-deductible under certain circumstances. Please consult with your tax advisor.

RETURN COMPLETED APPLICATION TO:

Association for Computing Machinery, Inc.
General Post Office
P.O. Box 30777
New York, NY 10087-0777

Questions? E-mail us at acmhelp@acm.org
Or call +1-800-342-6626 to speak to a live representative

Satisfaction Guaranteed!

payment:

Payment must accompany application. If paying by check or money order, make payable to ACM, Inc. in US dollars or foreign currency at current exchange rate.

❑ Visa/MasterCard ❑ American Express ❑ Check/money order

❑ Professional Member Dues ($99 or $198) $ _____

❑ ACM Digital Library ($99) $ _____

❑ Student Member Dues ($19, $42, or $62) $ _____

Total Amount Due $ _____

Card # Expiration date

Signature

DOI:10.1145/1506409.1506422

Article development led by acmqueue
queue.acm.org

What can be done to make Web browsers secure while preserving their usability?

BY THOMAS WADLOW AND VLAD GORELIK

Security in the Browser

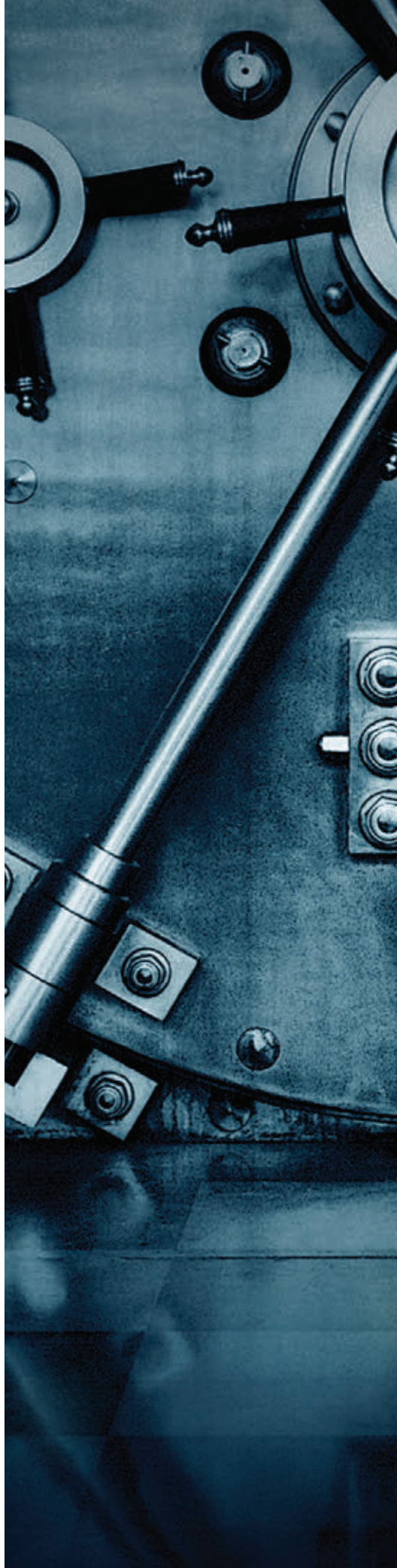


ILLUSTRATION BY JONATHAN BARKAT

"SEALED IN A depleted uranium sphere at the bottom of the ocean." That's the oft-quoted description of what it takes to make a computer reasonably secure. Obviously, in the Internet age or any other, such a machine would be fairly useless as well.

We live in interesting times. That computer on your desktop embodies the contradiction that faces a security engineer in the 21st century. It must be kept safe; and a lot of time, effort, and money is spent attempting to do exactly that. Firewalls are built to separate that machine from the Internet. Security audits tell us what programs must be deleted and what permissions changed so that the machine cannot be compromised. Virus checkers test all new software loaded on the machine for malicious content.

And yet, to make that fortress useful to us we demand that holes be chopped through the walls to permit us to run a Web browser. We complain if that browser is not given enough access to the rest of the computer. We insist on ease of use and speed, even if it makes all of our other defenses meaningless. And in many cases, we use browsers downloaded from the Internet without precaution, and configured by the owner of the desktop who has no security training or interest.

Browsers are at the heart of the Internet experience, and as such they are also at the heart of many of the security problems that plague users and developers alike.

The Use Model is Evolving...

Key features of early browsers included encryption and cookies, which were fine for the simple uses of the day. These techniques enabled the start of e-commerce, and monetizing the Web was what brought in the rest of the problems. Attackers who want money go where the money is, and there is money to be had on the Web.

Today, users expect far more from a browser. It should be able to handle sophisticated banking and shopping systems, display a wide variety of media, including video, audio, and animation, interact with the network on a micro scale (such as what happens when you move the cursor over a DVD selection in Netflix and see a summary of the movie), and update in as close to real time as possible—all without divulging sensitive information to bad guys or opening the door for attackers.

Consider AJAX, also known as Asynchronous JavaScript and XML. A Web page can contain code that establishes a network connection back to a server and conducts a conversation with that server that might bypass any number of security mechanisms integrated into the browser. The growing popularity of AJAX as a user-interface technique means an enterprise network often allows these connections, so that popular sites can function correctly.

The underlying mechanism of AJAX (which, despite the name, may not necessarily use JavaScript, XML, or be Asynchronous), is a function called XMLHttpRequest,[6] originally introduced by Microsoft for Internet Explorer, but now supported by Firefox, Safari, Opera and others. XMLHttpRequest allows a part of a Web page to make what is effectively a remote procedure call to a server across the Internet and use the results of that call in the context of the Web page. It is a powerful tool, but one that is open to a number of attacks.[2]

Flash, JavaScript, and Java all allow programs written by unknown third parties to run within the browser. Yes, there are sandboxes and safeguards, but as any attacker will tell you, a big step toward penetration is getting the target machine to run your code.

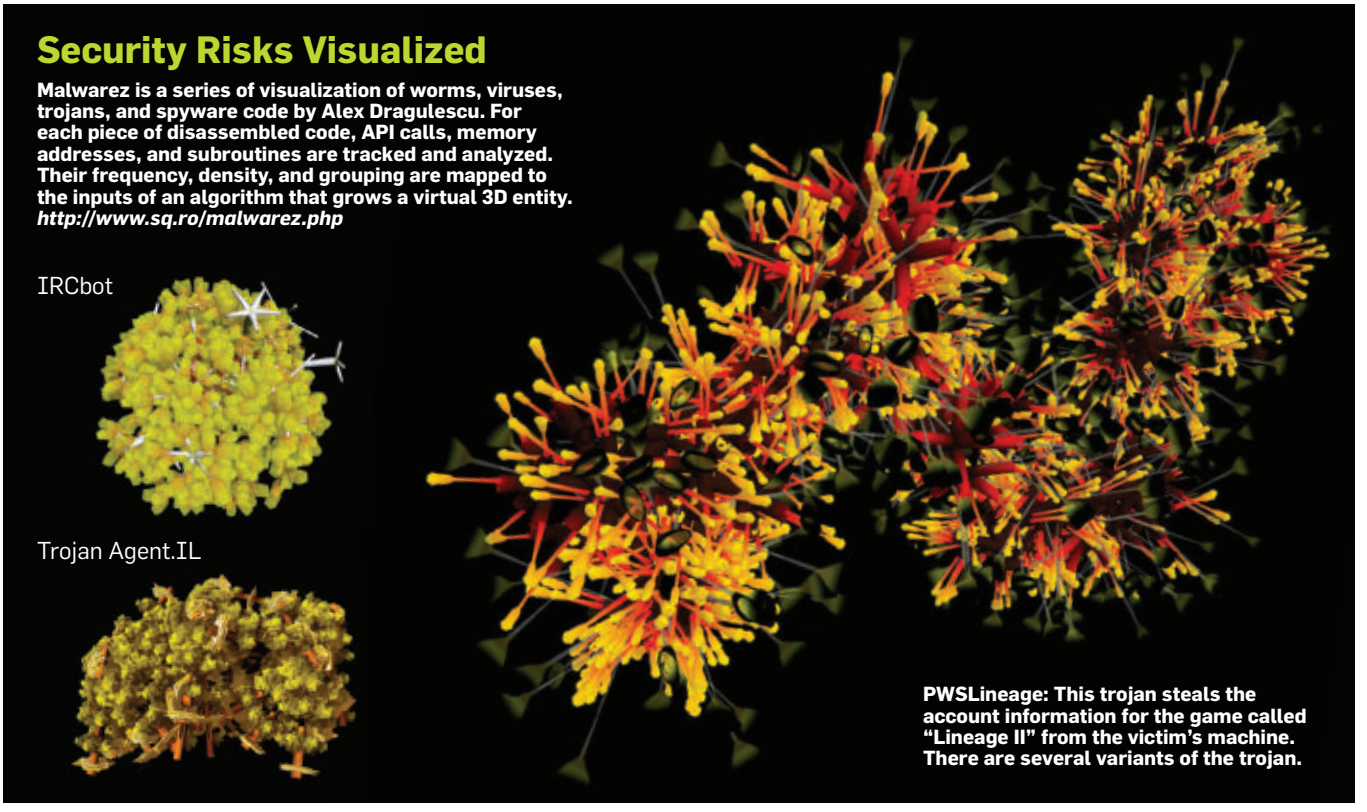
...And So Is The Threat Model

Early browsers had several major and noteworthy vulnerabilities, but they also had fewer types of attackers. The early attackers tended to be motivated by curiosity or scoring points with their peer groups. Modern browsers must defend against increasingly well-organized criminals who are looking for ways to turn browser vulnerabilities into money. They are aggressive, methodical, and willing to try a variety of attacks to see what works. And then there are those who work in gray areas, not quite violating the law, but pushing the envelope as much as possible to make a few dollars.

With more aggressive threats come more aggressive defenders. Security experts wanting to make names for themselves can release vulnerability information about browsers faster than browser developers may be prepared to react. While the roots of this type of disclosure

Security Risks Visualized

Malwarez is a series of visualization of worms, viruses, trojans, and spyware code by Alex Dragulescu. For each piece of disassembled code, API calls, memory addresses, and subroutines are tracked and analyzed. Their frequency, density, and grouping are mapped to the inputs of an algorithm that grows a virtual 3D entity. ***http://www.sq.ro/malwarez.php***



PWSLineage: This trojan steals the account information for the game called "Lineage II" from the victim's machine. There are several variants of the trojan.

are often driven by noble motives, the results can be devastating if they are not handled properly by all parties.

The flip side of early disclosure is the *zero-day exploit*. In this type of attack, an attacker learns of a flaw in a browser and moves to exploit it and profit from it before the security community has an opportunity to mount a defense.

Injection attacks (sometimes known as *cross-site scripting*, XSS) are when an attacker embeds commands or code in an otherwise legitimate Web request. This might include embedded SQL commands, stack-smashing attempts, in which data is crafted to exploit a programming vulnerability in the command interpreter, HTML injection, in which a post by a user (such as a comment in a blog) contains code intended to be executed by a viewer of that post.

Cross-site reference forgery (XSRF) is similar to XSS but it basically steals your cookie from another tab within your browser. This is relatively new, since tabbed browsing has only become popular in the last few years. It's an interesting demonstration of how a browser feature sometimes amplifies old problems. One of the reasons Google engineers implemented each tab in a separate process in Chrome was to avoid XSRF attacks.

A similarly named but different attack is the *cross-site request forgery*, in which, for example, the victim loads an HTML page that references an image whose `src` has been replaced by a call to another Web site, perhaps one that the victim has an account on. Variations of this attack include such things as mapping networks within the victim's enterprise for later use by another attack.

Add to this threats that are more social and less technical in nature—phishing,[5] for example, where a victim might receive a perfectly reasonable email message from a company that he does business with containing a link to a Web site that appears to be legitimate as well. He logs in, and the fake Web site snatches his username and password, which is then used for much less legitimate purposes than he would care for. A phishing scam depends much more on the gullibility of the user than the technology of the browser, but browsers often take much of the blame.

There are attacks of this nature based on the mistyping or misidentification of characters in a host name. A simple example of this would be that it is tricky to spot the difference between "google.com" and "googIe.com" (where the lowercase "L" has been replaced by an uppercase "I") in the sans-serif font so frequently used by browser URL entry fields. Expand that attack to Unicode and internationalization and you have something very painful and difficult to defend against.

The browser designer faces the Goldilocks problem. Either the porridge is too cold (not usable due to the demands of the security lockdown), or too hot (easy to abuse because not enough security measures are in place, or are too weak). Designing a configuration that is "just right" is nearly impossible because of evolving threats, uncovered bugs, and differing user tolerances for frustration.

Cookies are a long-used mechanism for storing information about a user or a session. They can be stolen, forged, poisoned, hijacked, or abused for denial-of-service attacks.[4] Yet, they remain an essential mechanism for many Web sites. Looking through the list of stored cookies on your browser can be very educational.

Similar to browser cookies are Flash Cookies. A regular HTTP cookie has a maximum size of 4KB and can usually be erased from a dialog box within the browser control panel. Flash Cookies, or Local Shared Objects (LSO)s are related to Adobe's Flash Player. They can store up to 100KB of data, have no expiration date, and normally cannot be deleted by the browser user, though some browser extensions are becoming available to assist with deleting them. Although Flash is run with a sandbox model, LSOs are stored on the user's disk and may be used in conjunction with other attacks.

In addition to Flash Cookies, the ActionScript language (how one writes a Flash application) supports XMLSockets that give Flash the ability to open network communication sessions. XMLSockets have some limitations—they aren't permitted to access ports lower than 1024 (where most system services reside), and they are allowed to connect only to the same subdomain where the originating Flash application resides. However, consider the case of a Flash game covertly run by an attacker. The attacker runs a high-numbered proxy on the same site, which can be accessed by XMLSockets from the victim's machine and redirected anywhere, for any purpose, bypassing XMLSocket limitations. This trick has already been used to unmask users who attempt to use anonymizing proxies to hide their identities.

Clickjacking is a relatively new attack, in which attackers present an apparently reasonable page, such as a Web game, but overlay on top of it a transparent page linked to another ser-

vice (such as the e-commerce interface for a store at which the victim has an account). By carefully positioning the buttons of the game, the attacker can cause the victim to perform actions from their store account without knowing that they've done so.

Security vs. Usability

Usability and security have long been at odds with each other in software design. The browser is no exception to that rule.

When browsing the Web or downloading files the user constantly needs to make choices about whether to trust a site or the content accessed from that site. Browser approaches to this have evolved over time—for example, browsers used to give a slight warning if you accessed a site with an invalid HTTPS certificate; now most browsers block sites with invalid certificates and make the user figure out how to unblock them. Similar approaches are taken with file downloads. Internet Explorer tends to ask the user several times before opening a downloaded file, especially if the file is not signed. Prompting the user for actions that are legitimate most of the time often creates user fatigue, which makes the user careless in walking the tightrope between software with a "reasonable but not excessive" security posture and a package that is either too open for safety or too closed to be useful. Most browsers today have evolved from the "make the user make the choice" model to the "block and require explicit override action" model.

In some cases the security of the browser has had a major impact on Web site design and usability. Browsers present a clear target for identity theft malware, since a lot of personal information flows through the browser at one time or another. This type of malware uses various techniques to steal users' credentials. One of these techniques is form grabbing—basically hooking the browser's internal code for sending form data to capture login information before it is encrypted by the SSL layer. Another technique is to log keyboard strokes to steal credentials when the user is typing information into a browser. These techniques have spawned various attempts by Web site designers to provide more advanced authentication methods, such as multifactor authentication with a hardware token and use of various click-based keyboards to avoid key loggers. In some cases some banks ask the user to authenticate each transaction with a hardware token. Although some of these techniques definitely improve security, they can place a pretty heavy burden on the end user.

Modern browsers must defend against increasingly well-organized criminals who are looking for ways to turn browser vulnerabilities into money. They are aggressive, methodical, and willing to try a variety of attacks to see what works.

Another usability feature of the Web browser that has been attacked by malware is the auto-complete functionality. Auto-complete saves the form information in a safe location and presents the user with options for what he typed before into a similar form. Several families of malware, such as the Goldun/Trojan Hearse, used this technique very effectively. The malware cracked the encrypted autocomplete data from the browser and send it back to the central server location without even having to wait for the user to log in to the site.

Given all the vulnerabilities out there and the willingness of attackers to exploit them, you might think that users would be clamoring for more security from their browsers. And some of them do...as long as it doesn't prevent any of their desired features from working.

Let's start with the browser software itself. From a security engineering perspective, the obvious choice for browser software (or any software) is to ship it in a "locked down" state, with all security features turned on, and let the user or enterprise weaken the security by enabling functions that they want. Consumer software that has done this has generally failed in the marketplace. Consumers want security, but they don't want to think about it or configure it. If the shipped configuration does what they want, they probably will not alter the configuration much, if at all.

So the browser designer faces the Goldilocks problem. Either the porridge is too cold (not usable because of the demands of the security lockdown) or too hot (too easy to abuse because not enough security measures are in place, or are too weak). Designing a configuration that is "just right" is nearly impossible because of evolving threats, uncovered bugs, and differing user tolerances for frustration.

There are a number of documents available that list steps one can take to lock down a Web browser. For example, one of those steps often is something like "Disable JavaScript." But few people actually ever do that—at least not

permanently, because using a browser with JavaScript turned off is annoying, and in many cases prevents you from visiting sites you have legitimate reasons to visit.

Cookies, while sometimes flushed to solve a problem, are essential to many Web sites, and having them disabled will prevent a wide range of services from working.

What is a Browser Designer To Do?

Browser developers have been working overtime to try and address some of these issues— and with some success— but it is definitely an uphill battle.

Proactive and reactive developers can generate an endless series of software updates. As a responsible defender, your dilemma is that allowing user these untested updates may break applications or even introduce security holes, but not allowing them may leave your enterprise open to even more serious attacks.

Distributed management provides some help in this area, but all major browsers are weaker than many defenders would like them to be. Microsoft provides the free Internet Explorer Administration Kit, which sets the bar for enterprise browser deployment and management tools, but that bar is lower than many would desire. FirefoxADM, an open source project for managing collections of Firefox browsers, is far more limited but a step in the right direction. FrontMotion provides a Web-based tool that allows a defender to create packages with approved software, configuration, and plugins for Firefox. All are available for the Windows platforms only.

Firefox and Google's Chrome browser have implemented "sandboxes," in which code run by the browser (such as JavaScript or Flash) is run in a compartmentalized area of the program that provides only limited resources for the program to run and whose design is heavily scrutinized for security flaws. Internet Explorer uses a zone-based security model, in which security features are enabled or disabled depending on what site is being accessed. Under Vista, it runs in what is known as Protected Mode, which limits the operating system privileges that the browser program can exercise.

However, open source developers must be especially careful about designing and implementing sandbox systems because their sandbox source code is available to the attacker for study and testing. This is, of course, no surprise to the sandbox developers and one reason why open source sandboxes tend to improve quickly.

Browser developers have come up with several ways to combat phishing attacks as well, primarily heuristics to detect an attempted visit to a fraudulent site, techniques to aggregate lists of and warn about known phishing sites, and augmentation of login security.

Injection attacks are most properly defended against at the server, but the victim will often be the browser user, not the server owner. Therefore, browsers may implement policies that hamper the injection attack by limiting where resources may be accessed from within a particular page.

Firefox has aggressively pursued a strategy of patching known vulnerabilities and generates updates regularly. Internet Explorer 7 is a significant improvement over Internet Explorer 6 in this regard, though many more known-but-unpatched vulnerabilities exist in IE 7 than in Firefox. Chrome seems to be emulating Firefox, though it lacks the mindshare of the other two at the moment so fewer eyeballs are looking critically at it for flaws.[3]

Some browser developers are employing and refining their system for detecting, reporting, and responding to security flaws. Mozilla.org, the support and development organization for Firefox, enlists open source developers to assist with code reviews and offers open bug tracking systems so that bugs can be reported and the follow-up tracked.

From a defender point-of-view, these efforts are a mixed blessing. Because browser software may be freely downloaded from the Internet by any user, all browsers are suspect. A prudent defender might hope that the browser is sufficiently rugged, but he cannot count on that fact. Desktop *nix systems and Mac OS X allow browser software to be run at a lower permission level than Windows often does, but that safeguard may be circumvented by other user-driven configuration changes.

Conclusion

From a network security perspective, a browser is essentially a somewhat controlled hole in your organization's firewall that leads to the heart of what it is you are trying to protect. While browser designers do try to limit what attackers can do from within a browser, much of the security relies far too heavily on the browser user, who often has other interests besides security. There are limits to what a browser developer can compensate for, and browser users will not always accept the constraints of security that a browser establishes.

As this issue gets more exposure, browser developers are cooperating to some degree to share strategies for defense. Google has published an excellent *Browser Security Handbook*[1] that compares various browser features and defenses.

Attack and defense strategies are coevolving, as are the use and threat models. As always, anybody can break into anything if they have sufficient skill, motivation and opportunity. The job of browser developers, network administrators, and browser users is to modulate those three quantities to minimize the number of successful attacks.

And that is a very big job indeed. C

Related articles on queue.acm.org

Criminal Code
Tom Wadlow and Vlad Gorelik
http://queue.acm.org/detail.cfm?id=1180192

Cybercrime: An Epidemic
Team Cymru
http://queue.acm.org/detail.cfm?id=1180190

Building Secure Web Applications
George Neville-Neil
http://queue.acm.org/detail.cfm?id=1281889

References

1. Google. *Browser Security Handbook*; http://code.google.com/p/browsersec/wiki/Main.
2. ISecPartners. *Attacking AJAX Applications*; http://www.isecpartners.com/files/iSEC-Attacking_AJAX_Applications.BH2006.pdf.
3. Wikipedia. Comparison of Web Browsers; http://en.wikipedia.org/wiki/Comparison_of_web_browsers.
4. Wikipedia. HTTP cookie; http://en.wikipedia.org/wiki/HTTP_cookie.
5. Wikipedia. Phishing; http://en.wikipedia.org/wiki/Phishing.
6. Wikipedia. XMLHttpRequest; http://en.wikipedia.org/wiki/XMLHttpRequest.

Thomas A. Wadlow is a network and computer security consultant, and the author of *The Process of Network Security*, Addison-Wesley Professional, 2000.

Vlad Gorelik is vice president of engineering at AVG Technologies where he heads up the development of behavioral malware detection and removal technologies. Previously he spent several years as CTO of Sana Security, leading the company's efforts in creating products to fight malware. He has multiple patents and filed patent applications in software technology and computer security.

DOI:10.1145/1506409.1506424

Article development led by acmqueue
queue.acm.org

Bad application programming interfaces plague software engineering. How do we get things right?

BY MICHI HENNING

API Design Matters

AFTER MORE THAN 25 years as a software engineer, I still find myself underestimating the time it takes to complete a particular programming task. Sometimes, the resulting schedule slip is caused by my own shortcomings: as I dig into a problem, I simply discover it is a lot more difficult than I initially thought, so the problem takes longer to solve—such is life as a programmer. Just as often I know exactly what I want to achieve and how to achieve it, but it still takes far longer than anticipated. When that happens, it is usually because I am struggling with an application programming interface (API) that seems to do its level best to throw rocks in my path and make my life difficult. What I find telling is that, even after 25 years of progress in software engineering, this still happens. Worse, recent APIs implemented in modern programming languages make the same mistakes as their 20-year-old counterparts written in C. There seems to be something elusive about API design that, despite years of progress, we have yet to master.

Good APIs are hard. We all recognize a good API when we get to use one. Good APIs are a joy to use. They work without friction and almost disappear

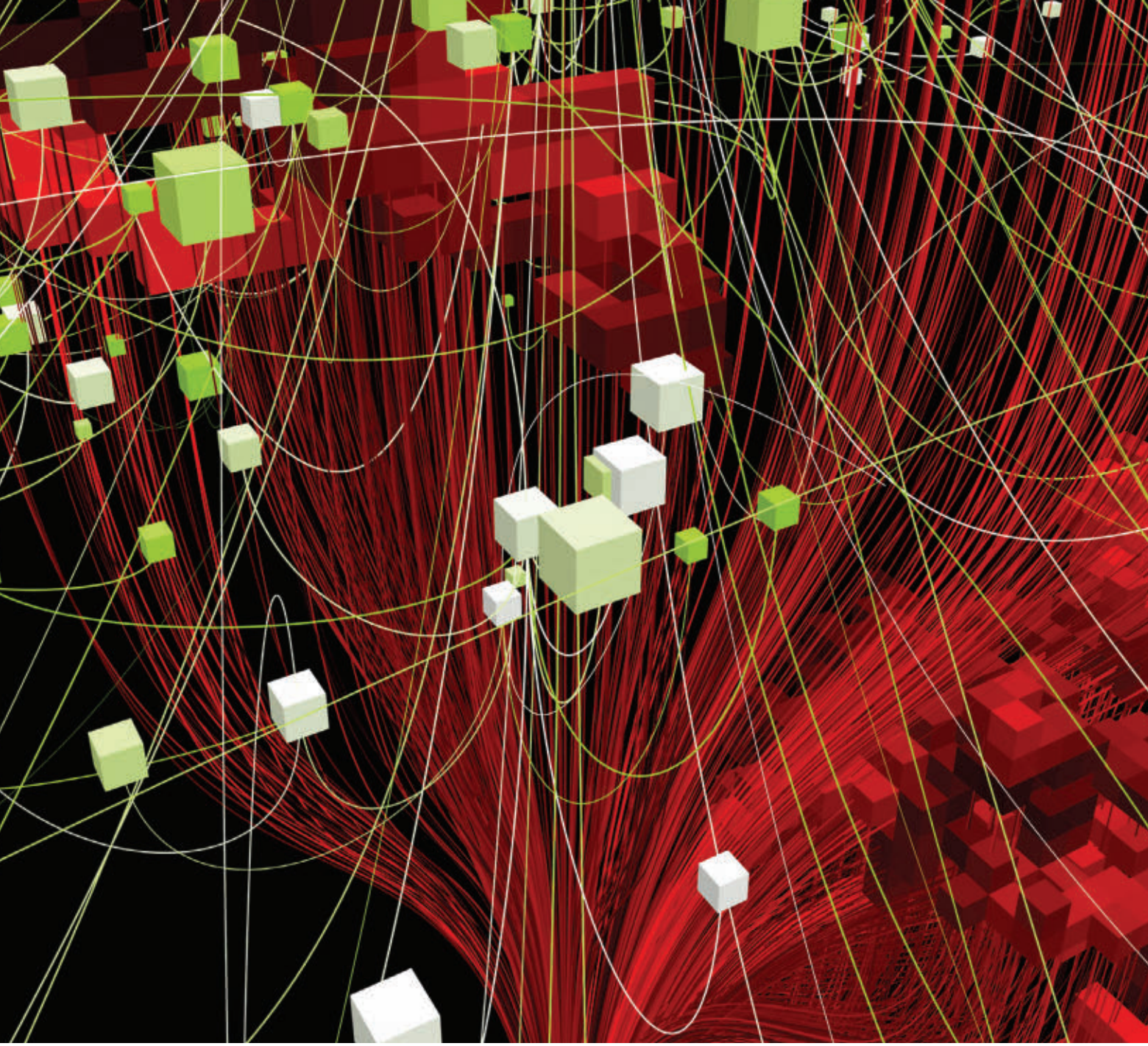


ILLUSTRATION BY LEANDER HERZOG

from sight: the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions.

So, why are there so many bad APIs around? The prime reason is that, for every way to design an API correctly, there are usually dozens of ways to design it incorrectly. Simply put, it is very easy to create a bad API and rather difficult to create a good one. Even minor and quite innocent design flaws have a tendency to get magnified out of all proportion because APIs are provided once, but are called many times. If a design flaw results in awkward or inefficient code, the resulting problems show up at every point the API is called. In addition, separate design flaws that in isolation are minor can interact with each other in surprisingly damaging ways and quickly lead to a huge amount of collateral damage.

Bad APIs are easy. Let me show you by example how seemingly innocuous design choices can have far-reaching ramifications. This example, which I came across in my day-to-day work, nicely illustrates the consequences of bad design. (Literally hundreds of similar examples can be found in virtually every platform; my intent is not to single out .NET in particular.)

Figure 1 shows the interface to the .NET socket `Select()` function in C#. The call accepts three lists of sockets that are to be monitored: a list of sockets to check for readability, a list of sockets to check for writeability, and a list of sockets to check for errors. A typical use of `Select()` is in servers that accept incoming requests from multiple clients; the server calls `Select()` in a loop and, in each iteration of the loop, deals with whatever sockets are ready before calling `Select()` again. This loop looks something like the one shown in Figure 1.

The first observation is that `Se-`

lect() overwrites its arguments: the lists passed into the call are replaced with lists containing only those sockets that are ready. As a rule, however, the set of sockets to be monitored rarely changes, and the most common case is that the server passes the same lists in each iteration. Because Select() overwrites its arguments, the caller must make a copy of each list before passing it to Select(). This is inconvenient and does not scale well: servers frequently need to monitor hundreds of sockets so, on each iteration, the code has to copy the lists before calling Select(). The cost of doing this is considerable.

A second observation is that, almost always, the list of sockets to monitor for errors is simply the union of the sockets to monitor for reading and writing. (It is rare that the caller wants to monitor a socket only for error conditions, but not for readability or writeability.) If a server monitors 100 sockets each for reading and writing, it ends up copying 300 list elements on each iteration: 100 each for the read, write, and error lists. If the sockets monitored for reading are not the same as the ones monitored for writing, but overlap for some sockets, constructing the error list gets harder because of the need to avoid placing the same socket more than once on the error list (or even more inefficient, if such duplicates are accepted).

Yet another observation is that Select() accepts a time-out value in microseconds: if no socket becomes ready within the specified time-out, Select() returns. Note, however, that the function has a void return type—that is, it does not indicate on return whether any sockets are ready. To determine whether any sockets are ready, the caller must test the length of all three lists; no socket is ready only if all three lists have zero length. If the caller happens to be interested in this case, it has to write a rather awkward test. Worse, Select() clobbers the caller's arguments if it times out and no socket is ready: the caller needs to make a copy of the three lists on each iteration even if nothing happens!

The documentation for Select() in .NET 1.1 states this about the time-out parameter: "The time to wait for a response, in microseconds." It offers no further explanation of the meaning of this parameter. Of course, the question immediately arises, "How do I wait indefinitely?" Seeing that .NET Select() is just a thin wrapper around the underlying Win32 API, the caller is likely to assume that a negative time-out value indicates that Select() should wait forever. A quick experiment, however, confirms that any time-out value equal to or less than zero is taken to mean "return immediately if no socket is ready." (This problem has been fixed in the .NET 2.0 version of Select().) To wait "forever," the best thing the caller can do is pass Int.MaxValue (2^{31}-1). That turns out to be a little over 35 minutes, which is nowhere near "forever." Moreover, how should Select() be used if a time-out is required that is not infinite, but longer than 35 minutes?

It is very easy to create a bad API and rather difficult to create a good one. Even minor and quite innocent design flaws have a tendency to get magnified out of all proportion because APIs are provided once, but are called many times.

When I first came across this problem, I thought, "Well, this is unfortunate, but not a big deal. I'll simply write a wrapper for Select() that transparently restarts the call if it times out after 35 minutes. Then I change all calls to Select() in the code to call that wrapper instead."

So, let's take a look at creating this drop-in replacement, called doSelect(), shown in Figure 2. The signature (prototype) of the call is the same as for the normal Select(), but we want to ensure that negative time-out values cause it to wait forever and that it is possible to wait for more than 35 minutes. Using a granularity of milliseconds for the time-out allows a time-out of a little more than 24 days, which I will assume is sufficient.

Note the terminating condition of the do-loop in the code in Figure 2: it is necessary to check the length of all three lists because Select() does not indicate whether it returned because of a time-out or because a socket is ready. Moreover, if the caller is not interested in using one or two of the three lists, it can pass either null or an empty list. This forces the code to use the awkward test to control the loop because, when Select() returns, one or two of the three lists may be null (if the caller passed null) or may be not null, but empty.

The problem here is that there are two legal parameter values for one and the same thing: both null and an emp-

ty list indicate that the caller is not interested in monitoring one of the passed lists. In itself, this is not a big deal but, if I want to reuse `Select()` as in the preceding code, it turns out to be rather inconvenient.

The second part of the code, which deals with restarting `Select()` for time-outs greater than 35 minutes, also gets rather complex, both because of the awkward test needed to detect whether a time-out has indeed occurred and because of the need to deal with the case in which milliseconds * 1000 does not divide `Int.MaxValue` without leaving a remainder.

We are not finished yet: the preceding code still contains comments in place of copying the input parameters and copying the results back into those parameters. One would think that this is easy: simply call a `Clone()` method, as one would do in Java. Unlike Java, however, .NET's type `Object` (which is the ultimate base type of all types) does not provide a `Clone` method; instead, for a type to be cloneable, it must explicitly derive from an `ICloneable` interface. The formal parameter type of the lists passed to `Select()` is `IList`, which is an interface and, therefore, abstract: I cannot instantiate things of type `IList`, only things derived from `IList`. The problem is that `IList` does not derive from `ICloneable`, so there is no convenient way to copy an `IList` except by explicitly iterating over the list contents and doing the job element by element. Similarly, there is no method on `IList` that would allow it to be easily overwritten with the contents of another list (which is necessary to copy the results back into the parameters before `doSelect()` returns). Again, the only way to achieve this is to iterate and copy the elements one at a time.

Another problem with `Select()` is that it accepts lists of sockets. Lists allow the same socket to appear more than once in each list, but doing so doesn't make sense: conceptually, what is passed are sets of sockets. So, why does `Select()`use lists? The answer is simple: the .NET collection classes do not include a set abstraction. Using `IList` to model a set is unfortunate: it creates a semantic problem because lists allow duplicates. (The behavior of `Select()` in the presence of duplicates is anybody's guess because it is not documented; checking against the actual behavior of the implementation is not all that useful because, in the absence of documentation, the behavior can change without warning.) Using `IList` to model a set is also detrimental in other ways: when a connection closes, the server must remove the corresponding socket from its lists. Doing so requires the server either to perform a linear search (which does not scale well) or to maintain the lists in sorted order so it can use a split search (which is more work). This is a good example of how design flaws have a tendency to spread and cause collateral damage: an oversight in one API causes grief in an unrelated API.

I will spare you the details of how to complete the wrapper code. Suffice it to say that the supposedly simple wrapper I set out to write, by the time I had added parameter copying, error handling, and a few comments, ran to nearly 100 lines of fairly complex code. All this because of a few seemingly minor design flaws:

- `Select()`overwrites its arguments.
- `Select()`does not provide a simple indicator that would allow the caller to distinguish a return because of a time-out from a return because a socket is ready.
- `Select()`does not allow a time-out longer than 35 minutes.
- `Select()`uses lists instead of sets of sockets.

Here is what `Select()` could look like instead:

```
public static int
Select(ISet checkRead,
       ISet checkWrite,
       Timespan seconds,
       out ISet readable,
       out ISet writeable,
       out ISet error);
```

With this version, the caller provides sets to monitor sockets for reading and writing, but no error set: sockets in both the read set and the write set are automatically monitored for errors. The time-out is provided as a `Timespan` (a type provided by .NET) that has resolution down to 100 nanoseconds, a range of more than 10 million days, and can be negative (or null) to cover the "wait forever" case. Instead of overwriting its arguments, this version returns the sockets that are ready for reading, writing, and have encountered an error as separate sets, and it returns the number of sockets

Figure 1: The .NET socket Select() in C++.

```
public static void Select(List checkRead, List checkWrite,
                          List checkError, int microseconds);
// Server code
int timeout = ...;
ArrayList readList = ...; // Sockets to monitor for reading.
ArrayList writeList = ...; // Sockets to monitor for writing.
ArrayList errorList; // Sockets to monitor for errors.

while (!done) {
    SocketList readTmp = readList.Clone();
    SocketList writeTmp = writeList.Clone();
    SocketList errorTmp = readList.Clone();
    Select(readTmp, writeTmp, errorTmp, timeout);
    for (int i = 0; i < readTmp.Count; ++i) {
      // Deal with each socket that is ready for reading...
    }
    for (int i = 0; i < writeTmp.Count; ++i) {
      // Deal with each socket that is redy for writing...
    }
    for (int i = 0; i < errorTmp.Count; ++i) {
      // Deal with each socket that encountered an error...
    }
    if (readTmp.Count == 0 &&
        writeTmp.Count == 0 &&
        errorTmp.Count == 0) {
          // No sockets are ready...
    }
}
```

that are ready or zero, in which case the call returned because the time-out was reached. With this simple change, the usability problems disappear and, because the caller no longer needs to copy the arguments, the code is far more efficient as well.

There are many other ways to fix the problems with `Select()` (such as the approach used by `epoll()`). The point of this example is not to come up with the ultimate version of `Select()`, but to demonstrate how a small number of minor oversights can quickly add up to create code that is messy, difficult to maintain, error prone, and inefficient. With a slightly better interface to `Select()`, none of the code I outlined here would be necessary, and I (and probably many other programmers) would have saved considerable time and effort.

Figure 2: The doSelect() function.

```
public void doSelect(List checkRead, List checkWrite,
                     List checkError, int milliseconds)
{
    ArrayList readCopy; // Copies of the three parameters because
    ArrayList writeCopy; // Select() clobbers them.
    ArrayList errorCopy;
    if (milliseconds <= 0) {
      // Simulate waiting forever.
      do {
          // Make copy of the three lists here...
          Select(readCopy, writeCopy, errorCopy, Int32.MaxValue);
      } while ((readCopy == null || readCopy.Count == 0) &&
               (writeCopy == null || writeCopy.Count == 0) &&
               (errorCopy == null || errorCopy.Count == 0));
    } else {
      // Deal with non-infinite timeouts.
      while ((milliseconds > Int32.MaxValue / 1000) &&
             (readCopy == null || readCopy.Count == 0) &&
             (writeCopy == null || writeCopy.Count == 0) &&
             (errorCopy == null || errorCopy.Count == 0)) {
          // Make a copy of the three lists here...
          Select(readCopy, writeCopy, errorCopy,
                 (Int32.MaxValue / 1000) * 1000);
          milliseconds -= Int32.MaxValue / 1000;
      }
    }
    if ((readCopy == null || readCopy.Count == 0) &&
        (writeCopy == null || writeCopy.Count == 0) &&
        (errorCopy == null || errorCopy == 0)) {
          Select(checkRead, checkWrite, checkError, milliseconds*1000);
    }
    // Copy the three lists back into the original parameters here...
}
```

The Cost of Poor APIs

The consequences of poor API design are numerous and serious. Poor APIs are difficult to program with and often require additional code to be written, as in the preceding example. If nothing else, this additional code makes programs larger and less efficient because each line of unnecessary code increases working set size and reduces CPU cache hits. Moreover, as in the preceding example, poor design can lead to inherently inefficient code by forcing unnecessary data copies. (Another popular design flaw—namely, throwing exceptions for expected outcomes—also causes inefficiencies because catching and handling exceptions is almost always slower than testing a return value.)

The effects of poor APIs, however, go far beyond inefficient code: poor APIs are harder to understand and more difficult to work with than good ones. In other words, programmers take longer to write code against poor APIs than against good ones, so poor APIs directly lead to increased development cost. Poor APIs often require not only extra code, but also more complex code that provides more places where bugs can hide. The cost is increased testing effort and increased likelihood for bugs to go undetected until the software is deployed in the field, when the cost of fixing bugs is highest.

Much of software development is about creating abstractions, and APIs are the visible interfaces to these abstractions. Abstractions reduce complexity because they throw away irrelevant detail and retain only the information that is necessary for a particular job. Abstractions do not exist in isolation; rather, we layer abstractions on top of each other. Application code calls higher-level libraries that, in turn, are often implemented by calling on the services provided by lower-level libraries that, in turn, call on the services provided by the system call interface of an operating system. This hierarchy of abstraction layers is an immensely powerful and useful concept. Without it, software as we know it could not exist because programmers would be completely overwhelmed by complexity.

The lower in the abstraction hierarchy an API defect occurs, the more serious are the consequences. If I mis-design a function in my own code, the only person affected is me, because I am the only caller of the function. If I mis-design a function in one of our project libraries, potentially all of my colleagues suffer. If I mis-design a function in a widely published library, potentially tens of thousands of programmers suffer.

Of course, end users also suffer. The suffering can take many forms, but the cumulative cost is invariably high. For example, if Microsoft Word contains a bug that causes it to crash occasionally because of a mis-designed API, thousands or hundreds of thousands of end users lose valuable time. Similarly, consider the numerous security holes in countless applications and system software that, ultimately, are caused by unsafe I/O and string manipulation functions in the standard C library (such as `scanf()` and `strcpy()`). The effects of these poorly designed APIs are still with us more than 30 years after they were created, and the cumulative cost of the design defects easily runs to many billions of dollars.

Perversely, layering of abstractions is often used to trivialize the impact of a bad API: "It doesn't matter—we can just write a wrapper to hide the problems." This argument could not be more wrong because it ignores the

cost of doing so. First, even the most efficient wrapper adds some cost in terms of memory and execution speed (and wrappers are often far from efficient). Second, for a widely used API, the wrapper will be written thousands of times, whereas getting the API right in the first place needs to be done only once. Third, more often than not, the wrapper creates its own set of problems: the .NET `Select()` function is a wrapper around the underlying C function; the .NET version first fails to fix the poor interface of the original, and then adds its own share of problems by omitting the return value, getting the time-out wrong, and passing lists instead of sets. So, while creating a wrapper can help to make bad APIs more usable, that does not mean that bad APIs do not matter: two wrongs don't make a right, and unnecessary wrappers just lead to bloatware.

A big problem with API documentation is that it is usually written after the API is implemented, and often written by the implementer.

How to do Better

There are a few guidelines to use when designing an API. These are not surefire ways to guarantee success, but being aware of these guidelines and taking them explicitly into account during design makes it much more likely that the result will turn out to be usable. The list is necessarily incomplete—doing the topic justice would require a large book. Nevertheless, here are a few of my favorite things to think about when creating an API.

An API must provide sufficient functionality for the caller to achieve its task. This seems obvious: an API that provides insufficient functionality is not complete. As illustrated by the inability of `Select()` to wait more than 35 minutes, however, such insufficiency can go undetected. It pays to go through a checklist of functionality during the design and ask, "Have I missed anything?"

An API should be minimal, without imposing undue inconvenience on the caller. This guideline simply says "smaller is better." The fewer types, functions, and parameters an API uses, the easier it is to learn, remember, and use correctly. This minimalism is important. Many APIs end up as a kitchen sink of convenience functions that can be composed of other, more fundamental functions. (The C++ standard string class with its more than 100 member functions is an example. After many years of programming in C++, I still find myself unable to use standard strings for anything nontrivial without consulting the manual.) The qualification of this guideline, without imposing undue inconvenience on the caller, is important because it draws attention to real-world use cases. To design an API well, the designer must have an understanding of the environment in which the API will be used and design to that environment. Whether or not to provide a nonfundamental convenience function depends on how often the designer anticipates that function will be needed. If the function will be used frequently, it is worth adding; if it is used only occasionally, the added complexity is unlikely to be worth the rare gain in convenience.

The Unix kernel violates this guideline with `wait()`, `waitpid()`, `wait3()`, and `wait4()`. The `wait4()` function is sufficient because it can be used to implement the functionality of the first three. There is also `waitid()`, which could almost, but not quite, be implemented in terms of `wait4()`. The caller has to read the documentation for all five functions in order to work out which one to use. It would be simpler and easier for the caller to have a single combined function instead. This is also an example of how concerns about backward compatibility erode APIs over time: the API accumulates crud that, eventually, does more damage than the good it ever did by remaining backward compatible. (And the sordid history of stumbling design remains for all the world to see.)

APIs cannot be designed without an understanding of their context. Consider a class that provides access to a set of name value pairs of strings, such as environment variables:

```
class NVPairs {
   public string
     lookup(string name);
   // ...
}
```

The `lookup` method provides access to the value stored by the named variable. Obviously, if a variable with the given name is set, the function re-

turns its value. How should the function behave if the variable is not set? There are several options:

- Throw a `VariableNotSet` exception.
- Return null.
- Return the empty string.

Throwing an exception is appropriate if the designer anticipates that looking for a variable that isn't there is not a common case and likely to indicate something that the caller would treat as an error. If so, throwing an exception is exactly the right thing because exceptions force the caller to deal with the error. On the other hand, the caller may look up a variable and, if it is not set, substitute a default value. If so, throwing an exception is exactly the wrong thing because handling an exception breaks the normal flow of control and is more difficult than testing for a null or empty return value.

Assuming that we decide not to throw an exception if a variable is not set, two obvious choices indicate that a lookup failed: return null or the empty string. Which one is correct? Again, the answer depends on the anticipated use cases. Returning null allows the caller to distinguish a variable that is not set at all from a variable that is set to the empty string, whereas returning the empty string for variables that are not set makes it impossible to distinguish a variable that was never set from a variable that was explicitly set to the empty string. Returning null is necessary if it is deemed important to be able to make this distinction; but, if the distinction is not important, it is better to return the empty string and never return null.

General-purpose APIs should be "policy-free;" special-purpose APIs should be "policy-rich." In the preceding guideline, I mentioned that correct design of an API depends on its context. This leads to a more fundamental design issue—namely, that APIs inevitably dictate policy: an API performs optimally only if the caller's use of the API is in agreement with the designer's anticipated use cases. Conversely, the designer of an API cannot help but dictate to the caller a particular set of semantics and a particular style of programming. It is important for designers to be aware of this: the extent to which an API sets policy has profound influence on its usability.

If little is known about the context in which an API is going to be used, the designer has little choice but to keep all options open and allow the API to be as widely applicable as possible. In the preceding lookup example, this calls for returning null for variables that are not set, because that choice allows the caller to layer its own policy on top of the API; with a few extra lines of code, the caller can treat lookup of a nonexistent variable as a hard error, substitute a default value, or treat unset and empty variables as equivalent. This generality, however, comes at a price for those callers who do not need the flexibility because it makes it harder for the caller to treat lookup of a nonexistent variable as an error.

This design tension is present in almost every API—the line between what should and should not be an error is very fine, and placing the line incorrectly quickly causes major pain. The more that is known about the context of an API, the more "fascist" the API can become—that is, the more policy it can set. Doing so is doing a favor to the caller because it catches errors that otherwise would go undetected. With careful design of types and parameters, errors can often be caught at compile time instead of being delayed until run time. Making the effort to do this is worthwhile because every error caught at compile time is one less bug that can incur extra cost during testing or in the field.

The `Select()` API fails this guideline because, by overwriting its arguments, it sets a policy that is in direct conflict with the most common use case. Similarly, the .NET `Receive()` API commits this crime for nonblocking sockets: it throws an exception if the call worked but no data is ready, and it returns zero without an exception if the connection is lost. This is the precise opposite of what the caller needs, and it is sobering to look at the mess of control flow this causes for the caller.

Sometimes, the design tension cannot be resolved despite the best efforts of the designer. This is often the case when little can be known about context because an API is low-level or must, by its nature, work in many different contexts (as is the case for general-purpose collection classes, for example). In this case, the strategy pattern can often be used to good effect. It allows the caller to supply a policy (for example, in the form of a caller-provided comparison function that is used to maintain ordered collections) and so keeps the design open. Depending on the programming language, caller-provided policies can be implemented with callbacks, virtual functions, delegates, or template parameters (among others). If the API provides sensible defaults, such externalized policies can lead to more flexibility without compromising usability and clarity. (Be careful, though, not to "pass the buck," as described later in this article.)

APIs should be designed from the perspective of the caller. When a programmer is given the job of creating an API, he or she is usually immediately in problem-solving mode: What data structures and algorithms do I need for the job, and what input and output parameters are necessary to get it done? It's all downhill from there: the implementer is focused on solving the problem, and the concerns of the caller are quickly forgotten. Here is a typical example of this:

```
makeTV(false, true);
```

This evidently is a function call that creates a TV. But what is the meaning of the parameters? Compare with the following:

```
makeTV(Color, FlatScreen);
```

The second version is much more readable to the caller: even without reading the manual, it is obvious that the call creates a color flat-screen TV. To the implementer, however, the first version is just as usable:

```
void makeTV(
    bool isBlackAndWhite,
    bool isFlatScreen)
{ /* ... */ }
```

The implementer gets nicely named variables that indicate whether the TV is black and white or color, and whether it has a flat screen or a conventional one, but that information is lost to the caller. The second version requires the implementer to do more work—

namely, to add enum definitions and change the function signature:

```
enum ColorType {
    Color,
    BlackAndWhite };
enum ScreenType {
    CRT,
    FlatScreen };
void makeTV(
    ColorType col,
    ScreenType st);
```

This alternative definition requires the implementer to think about the problem in terms of the caller. However, the implementer is preoccupied with getting the TV created, so there is little room in the implementer's mind for worrying about somebody else's problems.

A great way to get usable APIs is to let the customer (namely, the caller) write the function signature, and to give that signature to a programmer to implement. This step alone eliminates at least half of poor APIs: too often, the implementers of APIs never use their own creations, with disastrous consequences for usability. Moreover, an API is not about programming, data structures, or algorithms—an API is a user interface, just as much as a GUI. The user at the using end of the API is a programmer—that is, a human being. Even though we tend to think of APIs as machine interfaces, they are not: they are human–machine interfaces.

What should drive the design of APIs is not the needs of the implementer. After all, the implementer needs to implement the API only once, but the callers of the API need to call it hundreds or thousands of times. This means that good APIs are designed with the needs of the caller in mind, even if that makes the implementer's job more complicated.

Good APIs don't pass the buck. There are many ways to "pass the buck" when designing an API. A favorite way is to be afraid of setting policy: "Well, the caller might want to do this or that, and I can't be sure which, so I'll make it configurable." The typical outcome of this approach is functions that take five or 10 parameters. Because the designer does not have the spine to set policy and be clear about what the API should and should not do, the API ends up with far more complexity than necessary. This approach also violates minimalism and the principle of "I should not pay for what I don't use": if a function has 10 parameters, five of which are irrelevant for the majority of use cases, callers pay the price of supplying 10 parameters every time they make a call, even when they could not care less about the functionality provided by the extra five parameters. A good API is clear about what it wants to achieve and what it does not want to achieve, and is not afraid to be upfront about it. The resulting simplicity usually amply repays the minor loss of functionality, especially if the API has well-chosen fundamental operations that can easily be composed into more complex ones.

There is also a belief that older programmers "lose the edge." That belief is mistaken in my opinion; older programmers may not burn as much midnight oil as younger ones, but that's not because they are old, but because they get the job done without having to stay up past midnight.

Another way of passing the buck is to sacrifice usability on the altar of efficiency. For example, the CORBA C++ mapping requires callers to fastidiously keep track of memory allocation and deallocation responsibilities; the result is an API that makes it incredibly easy to corrupt memory. When benchmarking the mapping, it turns out to be quite fast because it avoids many memory allocations and deallocations. The performance gain, however, is an illusion because, instead of the API doing the dirty work, it makes the caller responsible for doing the dirty work—overall, the same number of memory allocations takes place regardless. In other words, a safer API could be provided with zero runtime overhead. By benchmarking only the work done inside the API (instead of the overall work done by both caller and API), the designers can claim to have created a better-performing API, even though the performance advantage is due only to selective accounting.

The original C version of `Select()` exhibits the same approach:

```
int select(int nfds,
    fd_set *readfds,
    fd_set *writefds,
    fd_set *exceptfds,
    struct timeval *timeout);
```

Like the .NET version, the C version also overwrites its arguments. This again reflects the needs of the implementer rather than the caller: it is easier and more efficient to clobber

the arguments than to allocate separate output arrays of file descriptors, and it avoids the problems of how to deallocate the output arrays again. All this really does, however, is shift the burden from implementer to caller—at a net efficiency gain of zero.

The Unix kernel also is not without blemish and passes the buck occasionally: many a programmer has cursed the decision to allow some system calls to be interrupted, forcing programmers to deal explicitly with `EINTR` and restart interrupted system calls manually, instead of having the kernel do this transparently.

Passing the buck can take many different forms, the details of which vary greatly from API to API. The key questions for the designer are: Is there anything I could reasonably do for the caller I am not doing? If so, do I have valid reasons for not doing it? Explicitly asking these questions makes design the result of a conscious process and discourages "design by accident."

APIs should be documented before they are implemented. A big problem with API documentation is that it is usually written after the API is implemented, and often written by the implementer. The implementer, however, is mentally contaminated by the implementation and will have a tendency simply to write down what he or she has done. This often leads to incomplete documentation because the implementer is too familiar with the API and assumes that some things are obvious when they are not. Worse, it often leads to APIs that miss important use cases entirely. On the other hand, if the caller (not the implementer) writes the documentation, the caller can approach the problem from a "this is what I need" perspective, unburdened by implementation concerns. This makes it more likely that the API addresses the needs of the caller and prevents many design flaws from arising in the first place.

Of course, the caller may ask for something that turns out to be unreasonable from an implementation perspective. Caller and implementer can then iterate over the design until they reach agreement. That way, neither caller nor implementation concerns are neglected.

Once documented and implemented, the API should be tried out by someone unfamiliar with it. Initially, that person should check how much of the API can be understood without looking at the documentation. If an API can be used without documentation, chances are that it is good: a self-documenting API is the best kind of API there is. While test driving the API and its documentation, the user is likely to ask important "what if" questions: What if the third parameter is null? Is that legal? What if I want to wait indefinitely for a socket to become ready? Can I do that? These questions often pinpoint design flaws, and a cross-check with the documentation will confirm whether the questions have answers and whether the answers are reasonable.

With the ever-growing importance of computing, there are APIs whose correct functioning is important almost beyond description.

Make sure that documentation is complete, particularly with respect to error behavior. The behavior of an API when things go wrong is as much a part of the formal contract as when things go right. Does the documentation say whether the API maintains the strong exception guarantee? Does it detail the state of out and in-out parameters in case of an error? Does it detail any side effects that may linger after an error has occurred? Does it provide enough information for the caller to make sense of an error? (Throwing a `DidntWork` exception from all socket operations just doesn't cut it!) Programmers *do* need to know how an API behaves when something goes wrong, and they *do* need to get detailed error information they can process programmatically. (Human-readable error messages are nice for diagnostics and debugging, but not nice if they are the only things available—there is nothing worse than having to write a parser for error strings just so I can control the flow of my program.)

Unit and system testing also have an impact on APIs because they can expose things that no one thought of earlier. Test results can help improve the documentation and, therefore, the API. (Yes, the documentation is part of the API.)

The worst person to write documentation is the implementer, and the worst time to write documentation is after implementation. Doing so greatly increases the chance that

interface, implementation, and documentation will *all* have problems.

Good APIs are ergonomic. Ergonomics is a major field of study in its own right, and probably one of the hardest parts of API design to pin down. Much has been written about this topic in the form of style guides that define naming conventions, code layout, documentation style, and so on. Beyond mere style issues though, achieving good ergonomics is hard because it raises complex cognitive and psychological issues. Programmers are humans and are not created with cookie cutters, so an API that seems fine to one programmer can be perceived as only so-so by another.

Especially for large and complex APIs, a major part of ergonomics relates to consistency. For example, an API is easier to use if its functions always place parameters of a particular type in the same order. Similarly, APIs are easier to use if they establish naming themes that group related functions together with a particular naming style. The same is true for APIs that establish simple and uniform conventions for related tasks and that use uniform error handling.

Consistency is important because not only does it make things easier to use and memorize, but it also enables transference of learning: having learned a part of an API, the caller also has learned much of the remainder of the API and so experiences minimal friction. Transference is important not only within APIs but also across APIs—the more concepts APIs can adopt from each other, the easier it becomes to master all of them. (The Unix standard I/O library violates this idea in a number of places. For example, the `read()` and `write()` system calls place the file descriptor first, but the standard library I/O calls, such as `fgets()` and `fputs()`, place the stream pointer last, except for `fscan()`and `fprint()`, which place it first. This lack of parallelism is jarring to many people.)

Good ergonomics and getting an API to "feel" right require a lot of expertise because the designer has to juggle numerous and often conflicting demands. Finding the correct trade-off among these demands is the hallmark of good design.

API Change Requires Cultural Change

I am convinced that it is possible to do better when it comes to API design. Apart from the nitty-gritty technical issues, I believe that we need to address a number of cultural issues to get on top of the API problem. What we need is not only technical wisdom, but also a change in the way we teach and practice software engineering.

Education. Back in the late 1970s and early 1980s, when I was cutting my teeth as a programmer and getting my degree, much of the emphasis in a budding programmer's education was on data structures and algorithms. They were the bread and butter of programming, and a good understanding of data structures such as lists, balanced trees, and hash tables was essential, as was a good understanding of common algorithms and their performance trade-offs. These were also the days when system libraries provided only the most basic functions, such as simple I/O and string manipulation; higher-level functions such as `bsearch()` and `qsort()` were the exception rather than the rule. This meant that it was de rigueur for a competent programmer to know how to write various data structures and manipulate them efficiently.

We have moved on considerably since then. Virtually every major development platform today comes with libraries full of pre-canned data structures and algorithms. In fact, these days, if I catch a programmer writing a linked list, that person had better have a very good reason for doing so instead of using an implementation provided by a system library.

Similarly, during this period, if I wanted to create software, I had to write pretty much everything from scratch: if I needed encryption, I wrote it from scratch; if I needed compression, I wrote it from scratch; if I needed inter-process communication, I wrote it from scratch. All this has changed dramatically with the open source movement. Today, open source is available for almost every imaginable kind of reusable functionality. As a result, the process of creating software has changed considerably: instead of creating functionality, much of today's software engineering is about integrating existing functionality or about repackaging it in some way. To put it differently: API design today is much more important than it was 20 years ago, not only because we are designing more APIs, but also because these APIs tend to provide access to much richer and more complex functionality.

Looking at the curriculum of many universities, it seems that this shift in emphasis has gone largely unnoticed. In my days as an undergraduate, no one ever bothered to explain how to decide whether something should be a return value or an out parameter, how to choose between raising an exception and returning an error code, or how to decide if it might be appropriate for a function to modify its arguments. Little seems to have changed since then: my son, who is currently working toward a software engineering degree at the same university where I earned my degree, tells me that still no one bothers to explain these things. Little wonder then that we see so many poorly designed APIs: it is not reasonable to expect programmers to be good at something they have never been taught.

Yet, good API design, even though complex, is something that can be taught. If undergraduates can learn how to write hash tables, they can also learn when it is appropriate to throw an exception as opposed to returning an error code, and they can learn to distinguish a poor API from a good one. What is needed is recognition of the importance of the topic; much of the research and wisdom are available already—all we need to do is pass them on.

Career Path. I am 49, and I write code. Looking around me, I realize how unusual this is: in my company, all of my programming colleagues are younger than I and, when I look at former programming colleagues, most of them no longer write code; instead, they have moved on to different positions (such as project manager) or have left the industry entirely. I see this trend everywhere in the software industry: older programmers are rare, quite often because no career path exists for them beyond a certain point. I recall how much effort it took me to resist a forced "promotion" into a management position at a former

company—I ended up staying a programmer, but was told that future pay increases were pretty much out of the question if I was unwilling to move into management.

There is also a belief that older programmers "lose the edge" and don't cut it anymore. That belief is mistaken in my opinion; older programmers may not burn as much midnight oil as younger ones, but that's not because they are old, but because they get the job done without having to stay up past midnight.

This loss of older programmers is unfortunate, particularly when it comes to API design. While good API design can be learned, there is no substitute for experience. Many good APIs were created by programmers who had to suffer under a bad one and then decided to redo the job, but properly this time. It takes time and a healthy dose of "once burned, twice shy" to gather the expertise that is necessary to do better. Unfortunately, the industry trend is to promote precisely its most experienced people away from programming, just when they could put their accumulated expertise to good use.

Another trend is for companies to promote their best programmers to designer or system architect. Typically, these programmers are farmed out to various projects as consultants, with the aim of ensuring that the project takes off on the right track and avoids mistakes it might make without the wisdom of the consultants. The intent of this practice is laudable, but the outcome is usually sobering: because the consultants are so valuable, having given their advice, they are moved to the next project long before implementation is finished, let alone testing and delivery. By the time the consultants have moved on, any problems with their earlier sage advice are no longer their problems, but the problems of a project they have long since left behind. In other words, the consultants never get to live through the consequences of their own design decisions, which is a perfect way to breed them into incompetence. The way to keep designers sharp and honest is to make them eat their own dog food. Any process that deprives designers of that feedback is ultimately doomed to failure.

External Controls. Years ago, I was working on a large development project that, for contractual reasons, was forced into an operating-system upgrade during a critical phase shortly before a delivery deadline. After the upgrade, the previously working system started behaving strangely and occasionally produced random and inexplicable failures. The process of tracking down the problem took nearly two days, during which a large team of programmers was mostly twiddling its thumbs. Ultimately, the cause turned out to be a change in the behavior of awk's `index()` function. Once we identified the problem, the fix was trivial—we simply installed the previous version of awk. The point is that a minor change in the semantics of a minor part of an API had cost the project thousands of dollars, and the change was the result of a side effect of a programmer fixing an unrelated bug.

This anecdote hints at a problem we will increasingly have to face in the future. With the ever-growing importance of computing, there are APIs whose correct functioning is important almost beyond description. For example, consider the importance of APIs such as the Unix system call interface, the C library, Win32, or OpenSSL. Any change in interface or semantics of these APIs incurs an enormous economic cost and can introduce vulnerabilities. It is irresponsible to allow a single company (let alone a single developer) to make changes to such critical APIs without external controls.

As an analogy, a building contractor cannot simply try out a new concrete mixture to see how well it performs. To use a new concrete mixture, a lengthy testing and approval process must be followed, and failure to follow that process incurs criminal penalties. At least for mission-critical APIs, a similar process is necessary, as a matter of self-defense: if a substantial fraction of the world's economy depends on the safety and correct functioning of certain APIs, it stands to reason that any changes to these APIs should be carefully monitored.

Whether such controls should take the form of legislation and criminal penalties is debatable. Legislation would likely introduce an entirely new set of problems, such as stifling innovation and making software more expensive. (The ongoing legal battle between Microsoft and the European Union is a case in point.) I see a real danger of just such a scenario occurring. Up to now, we have been lucky, and the damage caused by malware such as worms has been relatively minor. We won't be lucky forever: the first worm to exploit an API flaw to wipe out more than 10% of the world's PCs would cause economic and human damage on such a scale that legislators would be kicked into action. If that were to happen, we would likely swap one set of problems for another one that is worse.

What are the alternatives to legislation? The open source community has shown the way for many years: open peer review of APIs and implementations has proven an extremely effective way to ferret out design flaws, inefficiencies, and security holes. This process avoids the problems associated with legislation, catches many flaws before an API is widely used, and makes it more likely that, when a zero-day defect is discovered, it is fixed and a patch distributed promptly.

In the future, we will likely see a combination of both tighter legislative controls and more open peer review. Finding the right balance between the two is crucial to the future of computing and our economy. API design truly matters—but we had better realize it before events run away with things and remove any choice. C

Related articles on queue.acm.org

The Rise and Fall of CORBA
Michi Henning
http://queue.acm.org/detail.cfm?id=1142044

APIs with an Appetite
(Kode Vicious column)
http://queue.acm.org/detail.cfm?id=1229903

From COM to Common
Greg Olsen
http://queue.acm.org/detail.cfm?id=1142043

Michi Henning (michi@zeroc.com) is chief scientist of ZeroC, where he's working on the design and implementation of Ice—ZeroC's next-generation middleware. He previously worked on CORBA as a member of the Object Management Group's architecture board and as an ORB implementer, consultant, and trainer. With Steve Vinoski, he wrote *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

DOI:DOI:10.1145/1506409.1506423

Article development led by acmqueue
queue.acm.org

Lacking proper browser support, what steps can we take to debug production AJAX code?

BY ERIC SCHROCK

Debugging AJAX in Production

THE JAVASCRIPT LANGUAGE has a curious history. What began as a simple tool to let Web developers add dynamic elements to otherwise static Web pages has since evolved into the core of a complex platform for delivering Web-based applications. In the early days, the language's ability to handle failure silently was seen as a benefit. If an image rollover failed, it was better to preserve a seamless Web experience than to present the user with unsightly error dialogs.

This tolerance of failure has become a central design principle of modern browsers, where errors are silently logged to a hidden error console. Even when users are aware of the console, they find only a modicum of information, under the assumption that scripts are small and a single message indicating file and line number should be sufficient to identify the source of a problem.

This assumption no longer holds true, however, as the proliferation of sophisticated AJAX applications has permanently changed the design center of the JavaScript environment.

Scripts are large and complex, spanning a multitude of files and making extensive use of asynchronous, dynamically instantiated functions. Now, at best, script execution failure results in an awkward experience. At worst, the application ceases to work or corrupts server-side state. Tacitly accepting script errors is no longer appropriate, nor is a one-line number and message sufficient to identify a failure in a complex AJAX application. Accordingly, the lack of robust error messages and native stack traces has become one of the major difficulties with AJAX development today.

The severity of the problem depends on the nature of the debugging environment. During development, engineers have nearly unlimited freedom. They can recreate problems at will, launch an interactive debugger, or quickly modify and deploy test code, providing the ability to form and test hypotheses rapidly in order to determine the root cause of a problem. Everything changes, however, once an application leaves this haven for the

Figure 1: Automatically handling exceptions.

(a)

```
function mySetTimeout(callback, timeout)
        {
                var wrapper = function () {
                        try {
                                callback();
                        } catch (e) {
                                myHandleException(e);
                        }
                };

                return (setTimeout(wrapper, timeout));
        }
```

(b)

```
function myAddEventListener(obj, event, callback, capture)
        {
                var wrapper = function (evt) {
                        try {
                                callback(evt);
                        } catch (e) {
                                myHandleException(e);
                        }
                };
                if (!obj.listeners)
                        obj.listeners = new Array();
                obj.listeners.push({
                        event: event,
                        wrapper: wrapper,
                        capture: capture,
                        callback: callback
                });
                obj.addEventListener(event, wrapper, capture);
        }
```

Browser support.

Browser	Event	Message	File	Line	Stack
Firefox 3.0.5	window.onerror	x	x[1]	x[1]	
	DOM exception	x	x	x	
	runtime exception	x	x	x	x
	user exception				x[2]
IE 7.0.5730.13	window.onerror	x	x	x	
	DOM exception	x			
	runtime exception	x			
Safari 3.2.1	window.onerror				
	DOM exception	x	x	x	
	runtime exception	x	x	x	
	user exception		x	x	
Chrome 1.0.154.36	window.onerror				
	DOM exception	x			
	runtime exception	x			
	user exception				
Opera 9.63	window.onerror				
	DOM exception	x			x[3]
	runtime exception	x			x
	user exception				x[3]

1. DOM errors in Firefox do not have explicit file and line numbers, but the information is contained within the message.
2. Arbitrary exceptions do not have stack traces in Firefox, but those that use the Error() constructor do.
3. Opera can be configured to generate stack traces for exceptions, but it is not enabled by default.

production environment. Problems can be impossible to reproduce outside the user's environment, and gaining access to a system for interactive debugging is often out of the question. Running test code, even without requiring downtime, can prove worse than the problem itself. For these environments, the ability to debug problems after the fact is a necessity. When a bug is encountered in production, enough information must be preserved such that the root cause can be accurately determined, and this information must be made available in a form that can be easily transported from the user to engineering.

Depending on the browser, JavaScript has a rich set of tools for identifying the bugs at the root of problems during the development phase. Tools such as Firebug, Venkman, and built-in DOM (document object model) inspectors are immensely valuable. As with most languages, however, things become more difficult in production. Ideally, we would like to be able to obtain a complete dump of the JavaScript execution context, but no browser can support such a feature in a safe or practical manner. This leaves error messages as our only hope. These error messages must provide sufficient context to identify the root cause of an issue, and they must be integrated into the application experience such that the user can manage streams of errors and understand how to get the required information to developers for further analysis.

The first step in this process is to provide a means for displaying errors within the application. Although it is tempting simply to rely on `alert()` and its simple pop-up message, the visual experience associated with that is quite jarring. Large amounts of text do not scale well to pop-ups, and a flurry of such errors can require repeatedly dismissing the dialogs in rapid succession—sometimes making forward progress impossible. Many frameworks provide built-in consoles for this purpose, but a very simple hidden DOM element that allows us to expand, collapse, clear, and hide the console does the job nicely. With this integrated console, we can catch and display errors that would normally be lost to the browser error console. On

most browsers, errors can be caught by a top-level `window.onerror()` handler that provides a browser-specific message, file, and line number.

Simply dumping these messages to a user-visible console represents a major step forward, but even an accurate message, file, and line number can be worthless when debugging a problem in an AJAX application. Unless the bug is a simple typographical error, we need to better understand the context in which the error was encountered.

Faced with an unexpected error, the next question is almost always: "How and why are we here?" If we're lucky, we can just look at the source code and make some educated guesses. The most common method of improving this process is through stack traces. The ability to generate stack traces is the hallmark of a robust programming environment, but unfortunately this is also one feature often overlooked. Stack traces are often viewed as too difficult to construct, too expensive to make available in production, or simply not worth the effort to implement. Because they are commonly viewed as something that's required only in exceptional circumstances, stack traces can often be expensive to calculate. As the complexity of a system grows and as asynchrony is employed to a larger extent, however, this view becomes less tenable. In a message-passing system, for example, the context in which the original message was enqueued is often more important than the context of the failure once the message has been dequeued. In an AJAX environment (where *asynchronous* was worthy of a spot in the acronym), the need for closures often makes the context in which they have been instantiated more useful than the closures themselves.

Sadly, JavaScript support for stack traces is sorely lacking. The browsers that do support stack traces make them available only via thrown exceptions, and most browsers don't provide them at all. Stack traces are never available within global handlers such as `window.onerror()`, as the arguments are defined by a DOM that optimizes for the lowest common denominator. A `window.onexception()` handler that's passed as an exception object would be a welcome addition.

When a bug is encountered in production, enough information must be preserved such that the root cause can be accurately determined, and this information must be made available in a form that can be easily transported from the user to engineering.

Instead, we're forced to catch all exceptions explicitly. On the surface, this seems like a daunting task—we don't want to wrap every piece of code in a try/catch block. In an AJAX application, however, all JavaScript code is executed in one of four contexts:

- Global context while loading scripts;
- From an event handler in response to user interaction;
- From a timeout or interval; or
- From a callback when processing an XMLHTTPRequest.

The first case we must defer to `window.onerror()`, but since it happens while scripts are loading, it would be hard for such bugs to escape development. For the remaining cases, we can automatically wrap callbacks in try/catch blocks through our own registration function as illustrated in Figure 1a.

The table here describes the information that is available from a global context and when catching particular types of exceptions for different browsers. The table demonstrates the limits of integrated browser support. Without reliable stack traces on every exception, we are forced to generate programmatic stack traces for better coverage. Thankfully, the semantics of the `arguments` object allows us to write a function to generate a programmatic stack trace as depicted in Figure 2.

A full implementation would provide a means for skipping uninteresting frames, including native stack traces (via a try/catch block), and providing a `toString()` method for converting the results. We don't have file and line numbers, but we do have function names and arguments. Sadly, the proliferation of anonymous functions in JavaScript makes it difficult to get the canonical name of a function. The `toString()` method can give us the source for a particular function, but when printing a stack trace we need a name. The only effective way to accomplish this is to search the global namespace of all objects while constructing a human-readable name for the function along the way. This seems expensive, but we need to print the stack trace only in case of error. Most functions are either in the global namespace, one level deep, or two levels deep in the prototype of a particu-

Figure 2: Generating stacks.

```
function myStack()
{
        var caller, depth;
        var stack = new Array();
        for (caller = arguments.callee, depth = 0;
            caller && depth < 12;
            caller = caller.caller, depth++) {
                var args = new Array();
                for (var i = 0; i < caller.arguments.length; i++)
                        args.push(caller.arguments[i]);
                stack.push({
                    caller: caller,
                    args: args
                });
        }
        this.stack = stack;
}
```

Figure 3: Wrapping asynchronous requests.

(a)

```
        function dosomething(a, b)
                {
                service.dosomething(a + b, function (ret, err) {
                        if (err)
                                throw (err);
                        process(ret);
                });
        }
```

(b)

```
function dispatch(func, args, callback)
        {
                var stack = new myStack();
                dodispatch(func, args, function (ret, err) {
                        try {
                                callback(ret, err);
                        } catch (e) {
                                e.linkedStack = stack;
                                myHandleException(e);
                        }
                });
        }
```

lar object. To get a function's name, we simply need to search the members of the window object, all of their children, and all children of their prototype objects. If we find a match, then we can construct the name using this lineage.

With the function name and the arguments, we can display a reasonable facsimile of a stack trace, even on browsers without native support for stack traces. One caveat, however, is that getting function names doesn't work with Internet Explorer 7. For reasons that are not well understood, global functions are not included when iterating over members of the window object.

Careful construction of the global exception handler allows us to handle both native browser and dynamically generated exceptions. Although having stack traces attached to our custom exceptions is useful, the true power of this mechanism is evident when dealing with asynchronous closures in a complex environment, particularly asynchronous XMLHTTPRequest objects. In a complicated AJAX application, all server activity must happen asynchronously; otherwise, the browser will hang while waiting for a response. A typical service model will look something like Figure 3a.

If an exception occurs in the `process()` function, then a wrapper embedded in the service implementation will catch the result and hand it off to our exception handler. But the stack trace will end at `process()`, when what we really want is the stack trace at the point when `dosomething()` was called. Because our stack traces are generated on demand and are inexpensive to assemble, we can achieve this by recording the stack trace before dispatching every asynchronous call and then chaining it to any caught exception. The global exception handler will print all members of the exception, displaying both stack traces in the process. Our core dispatch routine would look something like Figure 3b. This allows transparent handling of server-side failures using the same exception handler. If an asynchronous closure generates an unanticipated exception, we can include the context in which the original XMLHTTPRequest was made.

By carefully following these design principles, we can construct an environment that dramatically improves our ability to debug issues by enabling users to provide developers with richer information that will allow for further analysis. Unfortunately, this environment is required to overcome the inadequacies of current JavaScript runtime environments. Without a single point to handle all uncaught exceptions, we are forced to wrap all callbacks in a try/catch block; and without reliable stack traces, we are forced to generate our own debugging infrastructure. It seems clear that a browser that implements these two features would soon become the preferred development environment for AJAX applications. Until that happens, careful design of the AJAX environment can still yield dramatic improvements in debuggability and serviceability for users of an application.

Related articles on queue.acm.org

Making the Move to AJAX (Case Study)
Jeff Norwalk
http://queue.acm.org/detail.cfm?id=1515744

Debugging in an Asynchronous World
Michael Donat
http://queue.acm.org/detail.cfm?id=945134

Debugging Devices
(Kode Vicious column)
http://queue.acm.org/detail.cfm?id=1483103

Eric Schrock has been a staff engineer at Sun Microsystems since 2003. After starting in the Solaris kernel group—where he worked on ZFS, among other things—Schrock spent the past few years helping to develop the Sun Storage 7000 series of appliances as part of the company's Fishworks engineering team.

The complete source code for the examples included here, as well as the latest version of the browser support table, can be found at http://blogs.sun.com/eschrock/resource/ajax/index.html.

DOI:10.1145/1506409.1506425

Multicore computers shift the burden of software performance from chip designers and processor architects to software developers.

BY JAMES LARUS

Spending Moore's Dividend

OVER THE PAST three decades, regular, predictable improvements in computers have been the norm, progress attributable to Moore's Law, the steady 40%-per-year increase in the number of transistors per chip unit area.

The Intel 8086, introduced in 1978, contained 29,000 transistors and ran at 5MHz. The Intel Core 2 Duo, introduced in 2006, contained 291 million transistors and ran at 2.93GHz.[9] During those 28 years, the number of transistors increased by 10,034 times and clock speed 586 times. This hardware evolution made all kinds of software run much faster. The Intel Pentium processor, introduced in 1995, achieved a SPECint95 benchmark score of 2.9, while the Intel Core 2 Duo achieved a SPECint2000 benchmark score of 3108.0, a 375-times increase in performance in 11 years.[a]

a Benchmarks from the 8080 era look trivial today and say little about modern processor performance. A realistic comparison over the decades requires a better starting point than the 8080. Moreover, the revision of the SPEC benchmarks every few years frustrates direct comparison. This comparison normalizes using the Dell Precision WorkStation 420 (800MHz PIII) that produced 364 SPECint2000 and 38.9 SPECint95, a ratio of 9.4.

These decades are also when the personal computer and packaged software industries were born and matured. Software development was facilitated by the comforting knowledge that every processor generation would run much faster than its predecessor. This assurance led to the cycle of innovation outlined in Figure 1. Faster processors enabled software vendors to add new features and functionality to software that in turn demanded larger development teams. The challenges of constructing increasingly complex software increased demand for higher-level programming languages and libraries. Their higher level of abstraction contributed to slower code and, in conjunction with larger and more complex programs, drove demand for faster processors and closed the cycle.

This era of steady growth of single-processor performance is over, however, and the industry has embarked on a historic transition from sequential to parallel computation. The introduction of mainstream parallel (multicore) processors in 2004 marked the end of a remarkable 30-year period during which sequential computer performance increased 40%–50% per year.[4] It ended when practical limits on power dissipation stopped the continual increases in clock speed, and a lack of exploitable instruction-level parallelism diminished the value of complex processor architectures.

Fortunately, Moore's Law has not been repealed. Semiconductor technology still doubles the number of transistors on a chip every two years.[7] However, this flood of transistors is now used to increase the number of independent processors on a chip, rather than to make an individual processor run faster.

The challenge the computing industry faces today is how to make parallel computing the mainstream method for improving software performance. Here, I look at this problem by asking how software consumed previous

Advanced Micro Devices multiple 45nm quad core die based on the Opteron processor, codenamed "Shanghai" (www.amd.com/)

PHOTOGRAPH COURTESY OF AMD

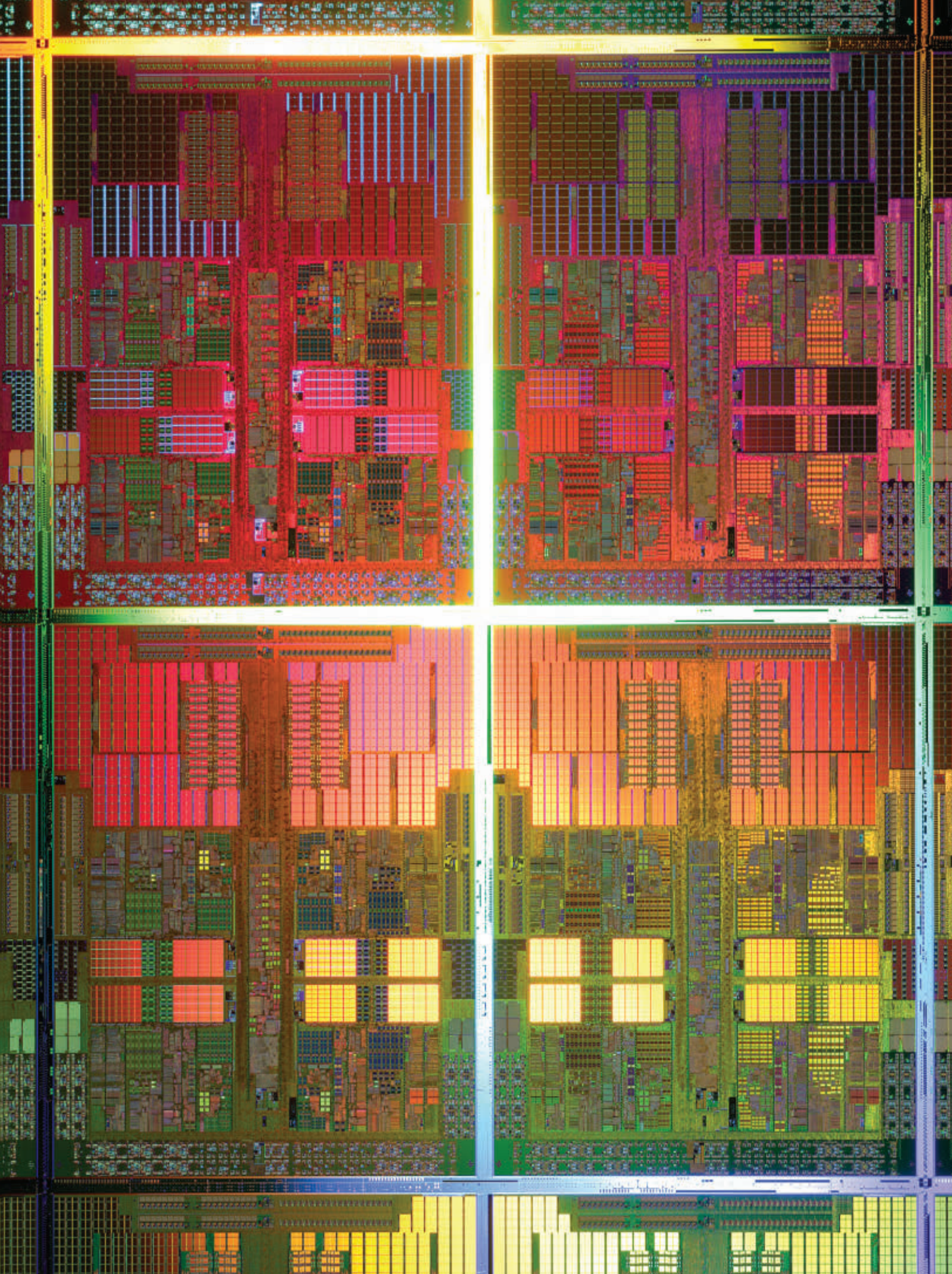


Figure 1: Cycle of innovation in the computer industry.

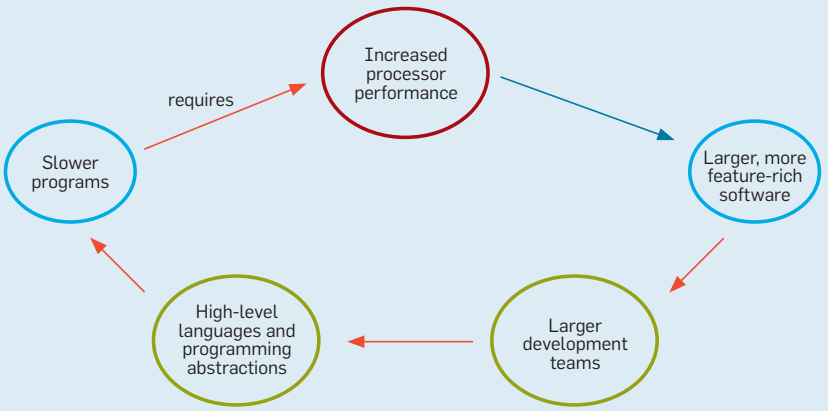
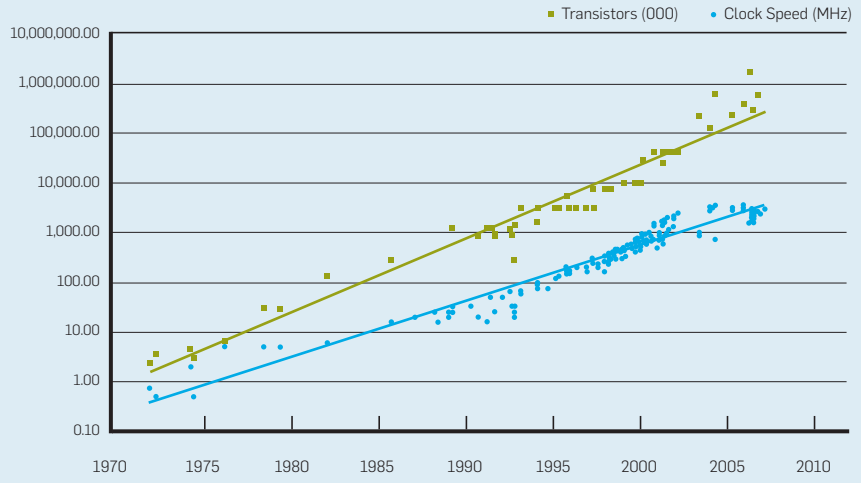


Figure 2: Improvement in Intel x86 processors; data from Olukotun,[18] Herb Sutter, a principal architect at Microsoft, and Intel.



processor-performance growth and whether multicore processors can satisfy the same needs. In short, how did we use the benefits of Moore's Law? Will parallelism continue the cycle of software innovation?

In 1965, Gordon Moore, a co-founder of Intel, postulated that the number of transistors that could be fabricated on a semiconductor chip would double every year,[17] a forecast he subsequently reduced to every second year.[10] Amazingly, this prediction still holds. Each generation of transistor is smaller and switches at a faster speed, allowing clock speed (and computer performance) to increase at a similar rate. Moreover, abundant transistors enabled architects to improve processor design by implementing sophisticated microarchitectures. For convenience, I call this combination of improvements in computers Moore's Dividend. Figure 2 depicts the evolution of Intel's x86 processors. The number of transistors in a processor increased at the rate predicted by Moore's Law, doubling every 24 months while clock frequency grew at a slightly slower rate.

These hardware improvements increased software performance. Figure 3 charts the highest SPEC integer benchmark score reported each month for single-processor x86 systems. Over a decade, integer processor performance increased by 52 times its former level.

Myhrvold's Laws

A common belief among software developers is that software grows at least at the same rate as the platform on which it runs. Nathan Myhrvold, former chief technology officer at Microsoft, memorably captured this wisdom with his four laws of software, following the premise that "software is a gas" due to its tendency to expand to fill the capacity of any computer (see the sidebar "Nathan Myhrvold's Four Laws of Software").

Support for this belief depends on the metric for the "volume" of software. Soon after Myhrvold published the "laws," the rate of growth of lines of code (LoC) in Windows diverged dramatically from the Moore's Law curve (see Figure 4). This makes sense intuitively; a software system might grow quickly in its early days, as basic functionality accrues, but exponential growth (such as the factor-of-four increase in lines of code between Windows 3.1 and Windows 95 over three years) is difficult to sustain without a similar increase in developer headcount or remarkable—unprecedented—improvement in software productivity.

Software volume is also measured in other ways, including necessary machine resources (such as processor speed, memory size, and capacity). Figure 5 outlines the recommended resources suggested by Microsoft for several versions of Windows. With the exception of disk space (which has increased faster than Moore's Law), the recommended configurations grew at roughly the same rate as Moore's Law.

How could software's resource requirements grow faster than its literal size (in terms of LoC)? Software changed and improved as computers became more capable. To most of the world, the real dividend of Moore's Law, and the reason to buy new computers, was this improvement, which enabled software to do more tasks and do them better than before.

How Was It Spent?

Determining where and how Moore's Dividend was spent is difficult for a number of reasons. Software evolves over a long period, but no one systematically measures changing resource consumption. It is possible to compare released systems, but many aspects of a system or application evolve between releases and without close investigation, and it is difficult to attribute visible differences to a particular factor. Here, I present some experimental hypotheses that await further research

to quantify their contributions to the overall computing experience:

Increased functionality. One of the clearest changes in software over the past 30 years has been a continually increasing baseline of expectations of what a personal computer can and should do. The changes are both qualitative and quantitative, but their cumulative effect has been steady growth in the computation needed to accomplish a specific task.

Software developers will tell you that improvement is continual and pervasive throughout the lifetime of software; new features extend it and, at the same time, raise its computational requirements. Consider the Windows print spooler, with a design that is still similar to Windows 95. Why does it not run 50 times faster today? Oliver Foehr,[5] a developer at Microsoft, analyzed it in 2008 and estimated the consequences of its evolution:

- Additional code over the years added new functionality, most notably improved security and notification, that affected performance by 1.5–4 times, depending on the scenario;
- Printer drivers added functionality for color management and improved treatment of text, graphics, and bookkeeping for a performance effect of a factor of 2;
- Printer resolution and color depth improved from 300*300 dpi at one bit per pixel to at least 600*600 dpi at 24 bits per pixel, or from 1MB to 96MB of image; and
- Memory latency and bandwidth did not keep up with processor speed; the spooler has poor locality due to large color lookup tables and graphics rendering, so its performance was slowed by the increased processor-memory gap.

Software rarely shrinks. Features are rarely removed, since it is difficult to ensure that no customers are still using them. Support for legacy compatibility ensures that the tide of resource requirements always rises and never recedes.

Large-scale, pervasive changes can affect overall system performance. Attacks of various sorts have led programmers to be more careful in writing low-level, pointer-manipulating code, forcing them to take extra care scrutinizing input data. Secure code requires more computation. One in-

Figure 3: Performance improvement in single-processor (x86) SPEC benchmarks (data from www.spec.org); the SPECint95 and SPECint2006 benchmark scores are normalized against SPECint2000.

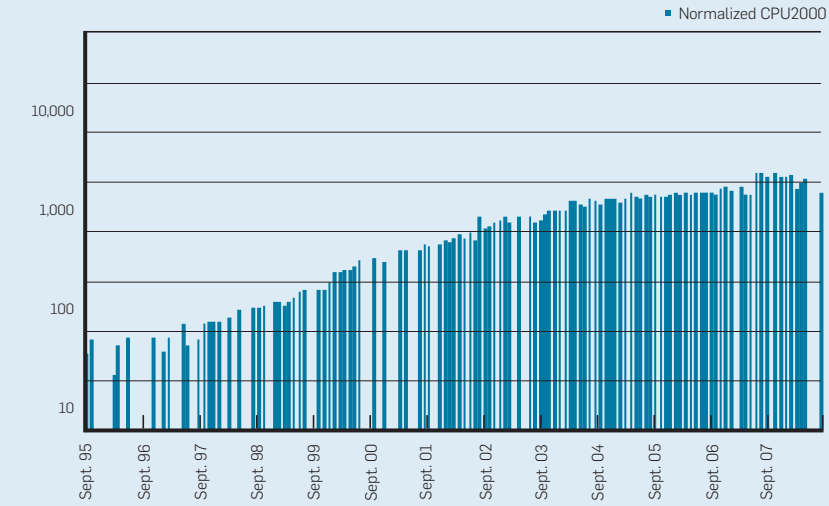


Figure 4: Windows code size (LoC) and Intel processor performance. Code size estimates are from various sources.[13–15]

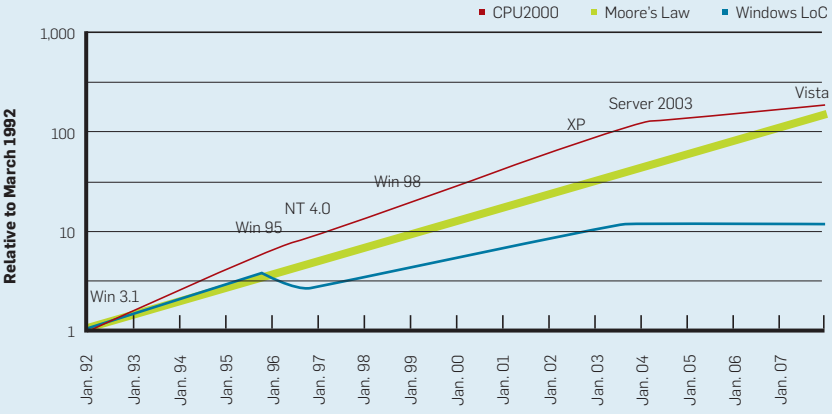


Figure 5: Recommended Windows configurations (maximum values from support.microsoft.com).

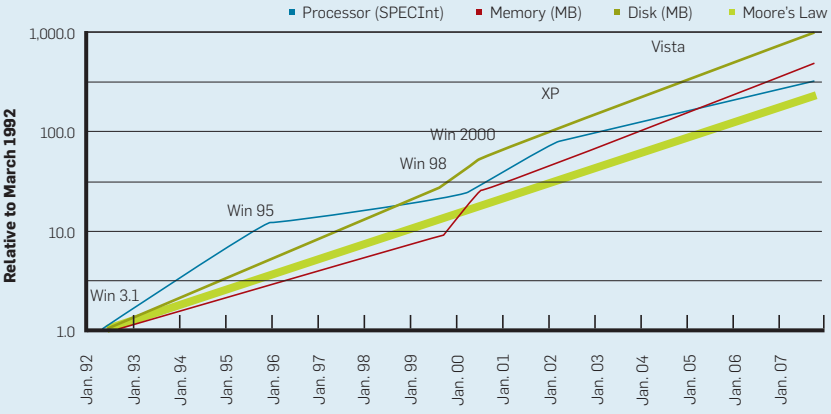


Table 1: Object-oriented complexity metrics (per binary); from an internal Microsoft Research document by Murphy, B. and Nagappan, N. Characterizing Vista Development, December 15, 2006.

	Vista/Win 2003 (Mean per binary)
Total Functions	1.45
Max Class Methods	1.22
Total Class Methods	1.59
Max Inheritance Depth	1.33
Total Inheritance Depth	1.54
Max Subclasses	3.87
Total Subclasses	2.27

dication is that array bounds and null pointer checks impose a time overhead of approximately 4.5% in the Singularity OS.[1] Also important, and equally difficult to measure, are the performance consequences of improved software-engineering practices (such as layering software architecture and modularizing systems to improve development and allow subsets).

Meanwhile, the data manipulated by computers is also evolving, from simple ASCII text to larger, structured objects (such as Word and Excel documents), to compressed documents (such as JPEG images), and more recently to space- and computation-inefficient formats (such as XML). The growing popularity of video introduces yet another format that is even more computationally expensive to manipulate.

Programming changes. Over the past 30 years, programming languages have evolved from assembly language and C code to increased use of higher-level languages. A major step was C++, which introduced object-oriented mechanisms (such as virtual-method dispatch). C++ also introduced abstraction mechanisms (such as classes and templates) that made possible rich libraries (such as the Standard Template Library). These language mechanisms required non-trivial, opaque runtime implementations that could be expensive to execute but improved software development through modularity, information hiding, and increased code reuse. In turn, these practices enabled the construction of ever-larger and more complex software.

Table 1 compares several key object-oriented complexity metrics between Windows 2003 and Vista, showing increased use of object-oriented features. For example, the number of classes per binary component increased 59% and the number of subclasses per binary 127% between the two systems.

These changes could have performance consequences. Comparing the SPEC CPU2000 and CPU2006 benchmarks, Kejariwal et al.[12] attributed the lower performance of the newer suite to increased complexity and size due to the inclusion of six new C++ benchmarks and enhancements to existing programs.

Safe, managed languages (such as C# and Java) further increased the level of programming by introducing garbage collection, richer class libraries (such as .NET and the Java Class Library), just-in-time compilation, and runtime reflection. All these features provide powerful abstractions for developing software but also consume memory and processor resources in nonobvious ways.

Language features can affect performance in two ways: The first is that a mechanism can be costly, even when not being used. Program reflection, a well-known example of a costly language feature, requires a runtime system to maintain a large amount of metadata on every method and class, even if the reflection features are not invoked. The second is that high-level languages hide details of a machine beneath a more abstract programming model. This leaves developers less aware of performance considerations and less able to understand and correct problems.

Mitchell et al.[16] analyzed the conversion of a date object in SOAP format to a Java Date object in IBM's Trade benchmark, a sample business application built on IBM Websphere. The conversion entailed 268 method calls and allocation of 70 objects. Jann et al.[11] analyzed this benchmark on consecutive implementations of IBM's POWER architecture, observing that "modern e-commerce applications are increasingly built out of easy-to-program, generalized but nonoptimized software components, resulting in substantive stress on the memory and storage subsystems of the computer."

I conducted simple programming experiments to compare the cost of implementing the archetypical Hello World program using various languages and features. Table 2 compares C and C# versions of the program, showing the latter has a working set 4.7–5.2 times larger. Another experiment measured the cost of displaying the string "Hello World" by both writing it to a console window and displaying it in a pop-up window. Table 3 shows that a dialog box is 20.7 times computationally more costly in C++ (using Microsoft Foundation Class) and 30.6 times more costly in C# (using Windows Forms). By comparison, the choice of language and runtime system made relatively little difference, as C# was only 1.5 times more costly than C++ for the console and 2.2 times more costly with a window.

This disparity is not a criticism of C#, .NET, or window systems; the

Table 2: Hello World benchmark running on Intel x86, Vista Enterprise, and Visual Studio 2008.

	Debug Build		Optimized Build	
Language	**Working Set**	**Startup Bytes**	**Working Set**	**Startup Bytes**
C	1,424K	6,162	1,304K	5,874
C++	6,756K	113,280	6,748K	87,62

Table 3: Execution cost of displaying "Hello World" string.

Mechanism	Timer Cycles (280ns)
C++, console	1,760
C++, window	36,375
C#, console	2.628
C#, window	80,348

overhead comes with a system that provides a much richer set of functionality that makes programming (and use) of computers faster and less error-prone. Moreover, the cost increases are far less than the performance improvement between the computers of the 1970s and 1980s—when C began—and today.

Decreased programmer focus. Abundant machine resources have allowed programmers to become complacent about performance and less aware of resource consumption in their code. Bill Gates 30 years ago famously changed the prompt in Altair Basic from "READY" to "OK" to save 5B of memory.[6] It is inconceivable today that a developer would be aware of such detail, let alone concerned about it, and rightly so, since a change of this magnitude is unnoticeable.

More significant, however, is a change in the developer mind-set that makes developers less aware of the resource requirements of the code they write:

Increased computer resources means fewer programs push the bounds of a computer's capacity or performance; hence many programs never receive extensive performance tuning. Donald Knuth's widely known dictum "premature optimization is the root of all evil" captures the typical practice of deferring performance optimization until code is nearly complete. When code performs acceptably on a baseline platform, it may still consume twice the resources it might require after further tuning. This practice ensures that many programs run at or near machine capacity and consequently helps guarantee that Moore's Dividend is fully spent at each new release;

Large teams of developers write software. The performance of a single developer's contribution is often difficult to understand or improve in isolation; that is, performance is not a modular property of software. Moreover, as systems become more complex, incorporate more feedback mechanisms, and run on less-predictable hardware, developers find it increasingly difficult to understand the performance consequences of their own decisions. A problem that is everyone's responsibility is no one's responsibility;

The performance of computers is increasingly difficult to understand. It used to suffice to count instructions alone to estimate code performance. As caches became more common, instruction and cache miss counts could identify program hot spots. However, latency-tolerant, out-of-order architectures require a far more detailed understanding of machine architecture to predict program performance; and

Programs written in high-level languages depend on compilers to achieve good performance. Compilers generate good code on average but are oblivious to major performance bottlenecks (such as disks and memory systems) and cannot fix fundamental flaws (such as bad algorithms).

This discussion is not a rejection of today's development practices. There is no way anyone could produce today's software using the artisan, handcraft practices that were possible and necessary for machines with 4K of memory. Moore's Dividend reduced the cost of running a program but increased the cost of developing one by encouraging ever-larger and more complex systems. Modern programming practices, starting with higher-level languages and rich libraries, counter this pressure by sacrificing runtime performance for reduced development effort.

Multicore and the Future

Anyone reading this is able to cite other scenarios in which Moore's Dividend was spent, but in the absence of further investigation and evidence, let's stop and examine the implications of these observations for future software and parallel computers:

Software evolution. Consider the normal process of software evolution, extension, and enhancement in sequential systems and applications. Sequential in this case excludes code running on parallel computers (such as databases, Web servers, scientific applications, and games) that presumably will continue to exploit parallelism on multicore processors.

Suppose a new product release adds functionality that uses a parallel algorithm to solve a computationally demanding task. Developing a parallel algorithm is a considerable challenge, but many problems (such as video processing, natural-language interaction, speech recognition, linear and nonlinear optimization, and machine learn-

Nathan Myhrvold's

Four Laws of Software

Nathan Myhrvold, a former astrophysicist, then Microsoft CTO, explained the dynamics of the computer and software industries as a natural consequence of his observation that software, like a gas, expands to fill its container (research.microsoft.com/acm97/nm/tsld026.htm) in the following ways:

SOFTWARE IS A GAS!
Windows NT lines of code (doubling time 866 days, growth rate 33.9% per year)
Browser Code Growth (doubling time 216 days, growth rate 221% per year)

SOFTWARE GROWS UNTIL IT BECOMES LIMITED BY MOORE'S LAW
Initial growth is quick, like gas expanding (like a browser)
Eventually limited by hardware (like NT)
Brings any processor to its knees, just before the new model is out

SOFTWARE GROWTH MAKES MOORE'S LAW POSSIBLE
That's why people buy new hardware, economic motivator
That's why chips get faster at the same price, not cheaper
Will continue as long as there is opportunity for new software

IMPOSSIBLE TO HAVE ENOUGH
New algorithms
New applications and new users
New notions of what is cool

ing) are computationally intensive. If computational speed inhibits adoption of these techniques—and parallel algorithms exist or can be developed—then multicore processors can enable the addition of compelling new functionality to applications.

Multicore processors are not a magic elixir, just another way to turn additional transistors into more performance. A problem solved with a multicore computer would also be solvable on a conventional processor—if sequential performance had continued its exponential increase. Moreover, multicore does not increase the rate of performance improvement, aside from one-time architectural shifts (such as replacing a single complex processor with a much larger number of simple cores).

New software features that successfully exploit parallelism differ from the evolutionary features added to most software written for conventional uniprocessor-based systems. A feature may benefit from parallelism if its computation is large enough to consume the processor for a significant amount of time, a characteristic that excludes incremental software improvements, small but pervasive software changes, and many simple program improvements.

Using parallel computation to implement a feature may not speed up an application as a whole due to Amdahl's Law's strict connection between the fraction of sequential execution and possible parallel speedup.[8] Eliminating sequential computation in the code for a feature is crucial, because even small amounts of serial execution can render a parallel machine ineffective.

An alternative use for multicore processors is to redesign a sequential application into a loosely coupled or asynchronous system in which computations run on separate processors. This approach uses parallelism to improve software architecture or responsiveness, rather than performance. For example, it is natural to separate monitoring and introspection features from program logic. Running these tasks on a separate processor can reduce perturbation of the mainline computation. Alternatively, extra processors can perform speculative computations to help minimize response time. These uses of parallelism are unlikely to scale with Moore's Law, but giving an application (or portions of an application) exclusive access to a set of processors might produce a more responsive system.

Applications that stop scaling with Moore's Law, either because they lack sufficient parallelism or because their developers no longer rewrite them, will be evolutionary dead ends.

Functionality that does not fit these patterns will not benefit from multicore; rather, such functionality will remain constrained by the static performance of a single processor. In the best case, the performance of a processor may continue to improve at a significantly slower rate (optimistic estimates range from 10% to 15% per year). But in some multicore chips, processors will run slower, as chip vendors simplify individual cores to lower power consumption and integrate more cores.

For many applications, most functionality is likely to remain sequential. For software developers to find the resources to add or change features, it may be necessary to eliminate old features or reduce their resource consumption. A paradoxical consequence of multicore is that sequential performance tuning and code-restructuring tools are likely to be increasingly important. Another likely consequence is that software vendors will be more aggressive in eliminating old or redundant features, making space for new code.

The regular growth in multicore parallelism poses an additional challenge to software evolution. Kathy Yelick, a professor of computer science at the University of California, Berkeley, has said that the experience of the high-performance computing community is that each decimal order of magnitude increase in parallelism requires a major redesign and rewrite of parallel code.[20] Multicore processors are likely to come into widespread use at the cusp of the first such change ($8 \rightarrow 16$); the next one ($64 \rightarrow 128$) is only three processor generations (six years) later. This observation is relevant only to applications that use scalable algorithms requiring large numbers of processors. Applications that stop scaling with Moore's Law, because they lack sufficient parallelism or their developers no longer rewrite them, are performance dead ends.

Parallelism will also force major changes in software development. Moore's Dividend enabled a shift to higher-level languages and libraries. The pressures driving this trend will not change, because increased abstraction helps improve security, reliability,

and program productivity. In the best case, parallelism enables new implementations of languages and features; for example, parallel garbage collectors reduce the pause time of computational threads, thereby enabling the use of safe languages in applications with real-time constraints.

Another approach that trades performance for productivity is to hide the underlying parallel implementation. Domain-specific languages and libraries can provide an implicitly parallel programming model that hides parallel programming from most developers, who instead use abstractions with semantics that do not change when running in parallel. For example, Google's MapReduce library utilizes a simple, well-known programming paradigm to initiate and coordinate independent tasks; equally important, it hides the complexity of running these tasks across a large number of computers.[3] The language and library implementers may struggle with parallelism, but other developers benefit from multicore without having to learn a new programming model.

Parallel software. Another major category of applications and systems already take advantage of parallelism; the two most notable examples are servers and high-performance computing, each providing different but important lessons to systems developers.

Servers have long been the main commercially successful type of parallel system. Their "embarrassingly parallel" workload consists of mostly independent requests that require little or no coordination and share little data. As such, it is relatively easy to build a parallel Web server application, since the programming model treats each request as a sequential computation. Building a Web site that scales well is an art; scale comes from replicating machines, which breaks the sequential abstraction, exposes parallelism, and requires coordinating and communicating across machine boundaries.

High-performance computing followed a different path that used parallel hardware because there was no alternative with comparable performance, not because scientific and technical computations are especially well suited to parallel solution. Parallel hardware is a tool for solving problems. The popular programming models—MPI and OpenMP—are performance-focused, error-prone abstractions that developers find difficult to use. More recently, game programming emerged as another realm of high-performance computing, with the same attributes of talented, highly motivated programmers spending great effort and time to squeeze the last bit of performance from complex hardware.[19]

If parallel programming is to be a mainstream programming model, it must follow the path of servers, not of high-performance computing. One alternative paradigm for parallel computing "Software as a Service" delivers software functionality across the Internet and revisits timesharing by executing some or all of an application on a shared server in the "cloud."[2] This approach to computing, like servers in general, is embarrassingly parallel and benefits directly from Moore's Dividend. Each application instance runs independently on a processor in a server. Moore's Dividend accrues directly to the service provider, even if the application is sequential. Each new generation of multicore processors halves the number of computers needed to serve a fixed workload or provide the headroom needed to add features or handle greater workloads. Despite the challenges of creating a new software paradigm and industry, this model of computation is likely to be popular, particularly for applications that do not benefit from multicore.

Conclusion

Moore's Dividend was spent in many ways and places, ranging from programming languages, models, architectures, and development practices, up through software functionality. Parallelism is not a surrogate for faster processors and cannot directly step into their roles. Multicore processors will change software as profoundly as previous hardware revolutions (such as the shift from vacuum tubes to transistors or transistors to integrated circuits) radically altered the size and cost of computers, the software written for them, and the industry that produced and sold the hardware and software. Parallelism will drive software in new directions (such as computationally intensive, game-like interfaces or services provided by the cloud) rather than continuing the evolutionary improvements made familiar by Moore's Dividend.

Acknowledgments

Many thanks to Al Aho (Columbia University), Doug Burger (Microsoft), David Callahan (Microsoft), Dennis Gannon (Microsoft), Mark Hill (University of Wisconsin), and Scott Wadsworth (Microsoft) for helpful comments and to Oliver Foehr (Microsoft) and Nachi Nagappan (Microsoft) for assistance with Microsoft data. C

References

1. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J.R. Deconstructing process isolation. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (San Jose, CA, Oct.). ACM Press, New York, 2006, 1–10.
2. Carr, N. *The Big Switch: Rewiring the World, From Edison to Google*. W.W. Norton, New York, 2008.
3. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.
4. Ekman, M., Warg, F., and Nilsson, J. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News 33*, 1 (Mar. 2005), 144–147.
5. Foehr, O. personal email communications, June 30, 2008.
6. Gates, B. Personal email (Apr. 10, 2008).
7. Hachman, M. Intel's Gelsinger predicts Intel Inside everything. *PC Magazine* (July 3, 2008).
8. Hill, M.D. and Marty, M.R. Amdahl's Law in the multicore era. *IEEE Computer 41*, 7 (July 2008), 33–38.
9. Intel. The Evolution of a Revolution. Santa Clara, CA, 2008; download.intel.com/pressroom/kits/IntelProcessorHistory.pdf.
10. Intel. *Excerpts from A Conversation with Gordon Moore: Moore's Law.* Video transcript, Santa Clara, CA, 2005; ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.
11. Jann, J., Burugula, R.S., Dubey, N., and Pattnaik, P. End-to-end performance of commercial applications in the face of changing hardware. *ACM SIGOPS Operating Systems Review 42*, 1 (Jan. 2008), 13–20.
12. Kejariwal, A., Hoflehner, G.F., Desai, D. Lavery, D.M., Nicolau, A., and Veidenbaum, A.V. Comparative characterization of SPEC CPU2000 and CPU2006 on Itanium architecture. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, June). ACM Press, New York, 2007, 361–362.
13. Lohr, S. and Markoff, J. Windows is so slow, but why? *New York Times* (Mar. 27, 2006); www.nytimes.com/2006/03/27/technology/27soft.html.
14. Maraia, V. *The Build Master: Microsoft's Software Configuration Management Best Practices.* Addison-Wesley, Upper Saddle River, NJ, 2006.
15. McGraw, G. *Software Security: Building Security In.* Addison-Wesley Professional, Boston, MA, 2006.
16. Mitchell, N., Sevitsky, G., and Srinivasan, H. The diary of a datum: An approach to analyzing runtime complexity in framework-based applications. In *Proceedings of the Workshop on Library-Centric Software Design* (San Diego, CA, Oct.). ACM Press, New York 2005, 85–90.
17. Moore, G.E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr. 1965), 56–59.
18. Olukotun, K. and Hammond, L. The future of microprocessors. *ACM Queue 3*, 7 (Sept. 2005), 26–29.
19. Sweeney, T. The next mainstream programming language: A game developer's perspective. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, SC, Jan.). ACM Press, New York, 2006, 269–269.
20. Yelick, K. AltTAB. Discussion on parallelism at Microsoft Research, Redmond, WA, July 19, 2006.

James Larus (larus@microsoft.com) is Director of Software Architecture in the Cloud Computing Futures project in Microsoft Research, Redmond, WA.

DOI:10.1145/1506409.1506426

The passage of time is essential to ensuring the repeatability and predictability of software and networks in cyber-physical systems.

BY EDWARD A. LEE

Computing Needs Time

MOST MICROPROCESSORS ARE embedded in systems that are not first-and-foremost computers. Rather, these systems are cars, medical devices, instruments, communication systems, industrial robots, toys, and games. Key to them is that they interact with physical processes through sensors and actuators. However, they increasingly resemble general-purpose computers, becoming networked and intelligent, often at the cost of dependability.

Even general-purpose computers are increasingly asked to interact with physical processes. They integrate media (such as video and audio), and through their migration to handheld platforms and pervasive computing systems, sense physical dynamics and control physical devices. They don't always do it well. The technological basis that engineers and computer scientists have chosen for general-purpose computing and networking does not support these applications well. Changes that ensure this support could improve them and enable many others.

The foundations of computing, rooted in Turing, Church, and von Neumann, are about the transformation of data, not physical dynamics. Computer scientists must rethink the core abstractions if they truly want to integrate computing with physical processes. That's why I focus here on a key aspect of physical processes—the passage of time—that is almost entirely absent in computing. This is not just about real-time systems, which accept the foundations and retrofit them with temporal properties. Although that technology has much to contribute to systems involving physical processes, it cannot solve the problem of computers functioning in the physical world alone because it is built on flawed technological foundations.

Many readers might object here. Computers are so fast that surely the passage of time in most physical processes is so slow it can be handled without special accommodation. But modern techniques (such as instruction scheduling, memory hierarchies, garbage collection, multitasking, and reusable component libraries that do not expose temporal properties in their interfaces) introduce enormous variability and unpredictability into computer-supported physical systems. These innovations are built on a key premise: that time is irrelevant to correctness and is at most a measure of quality. Faster is better, if you are willing to pay the price in terms of power consumption and hardware. By contrast, what these systems need is not faster computing but physical actions taken at the right time. Timeliness is a semantic property, not a quality factor.

But surely the "right time" is expecting too much, you might say. The physical world is neither precise nor reliable, so why should we demand such properties from computing systems? Instead, these systems must be robust and adaptive, performing reliably, despite being built out of unreliable components. While I agree that systems must be designed to be robust, we should not blithely discard the reliability we have. Electronics technology is astonishingly precise and reliable,

more than any other human invention ever made. Designers routinely deliver circuits that perform a logical function essentially perfectly, on time, billions of times per second, for years on end. Shouldn't we aggressively exploit this remarkable achievement?

We have been lulled into a false sense of confidence by the considerable success of embedded software in, say, automotive, aviation, and robotics applications. But the potential is much greater; hardware and software design has reached a tipping point, where computing and networking can indeed be integrated into the vast majority of artifacts made by humans. However, as we move to more networked, complex, intelligent applications, the problems of real-world compatibility and coordination are going to get worse. Embedded systems will no longer be black boxes, designed once and immutable in the field; they will be pieces of larger systems, a dance of electronics, networking, and physical processes. An emerging buzzword for such systems is cyber-physical systems, or CPS.

The charter for the CPS Summit in April 2008 (ike.ece.cmu.edu/twiki/bin/view/CpsSummit/WebHome) says "The integration of physical systems and processes with networked computing has led to the emergence of a new generation of engineered systems: cyber-physical systems. Such systems use computations and communication deeply embedded in and interacting with physical processes to add new capabilities to physical systems. These cyber-physical systems range from miniscule (pacemakers) to large-scale (the national power grid). Because computer-augmented devices are everywhere, they are a huge source of economic leverage.

"...it is a profound revolution that turns entire industrial sectors into producers of cyber-physical systems. This is not about adding computing and communication equipment to conventional products where both sides maintain separate identities. This is about merging computing and networking with physical systems to create new revolutionary science, technical capabilities and products."

The challenge of integrating computing and physical processes has been recognized for years,[20] motivating the emergence of hybrid systems theories. However, progress is limited to relatively simple systems combining ordinary differential equations with automata. Needed now are new breakthroughs in modeling, design, and analysis of such integrated systems.

CPS applications, arguably with the potential to rival the 20th century IT revolution, include high-confidence medical devices, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control (electric power, water resources, and communications systems), distributed robotics (telepresence, telemedicine), defense systems, manufacturing, and smart structures. It is easy to envision new capabilities that are technically well within striking distance but that would be extremely difficult to deploy with today's methods. Consider a city without traffic lights, where each car gives its driver adaptive information on speed limits and clearances to pass through intersections. We have all the technical pieces for such a system, but achieving

the requisite level of confidence in the technology is decades off.

Other applications seem inevitable but will be deployed without benefit of many (or most) developments in computing. For example, consider distributed real-time games that integrate sensors and actuators to change the (relatively passive) nature of online social interaction.

Today's computing and networking technologies unnecessarily impede progress toward these applications. In a 2005 article on "physical computing systems," Stankovic et al.[25] said "Existing technology for RTES [real-time embedded systems] design does not effectively support development of reliable and robust embedded systems." Here, I focus on the lack of temporal semantics. Today's "best-effort" operating system and networking technologies cannot produce the precision and reliability demanded by most of these applications.

Embedded systems will no longer be black boxes, designed once and immutable in the field; they will be pieces of larger systems, a dance of electronics, networking, and physical processes.

Glib Responses

Calling for a fundamental change in the core abstractions of computing is asking a lot of computer science. You may say that the problems can be addressed without such a revolution. To illustrate that a revolution is needed, I examine popular but misleading aphorisms, some suggesting that incremental changes will suffice:

Computing takes time. This brief sentence might suggest that if only software designers would accept this fact of life, then the problems of CPS could be dealt with. The word "computing" refers to an abstraction of a physical process that takes time. Every abstraction omits some detail (or it wouldn't be an abstraction), and one detail that computing omits is time. The choice to omit time has been beneficial to the development of computer science, enabling very sophisticated technology. But there is a price to pay in terms of predictability and reliability. This choice has resulted in a mismatch with many applications to which computing is applied. Asking software designers to accept the fact that computing takes time is the same as asking them to forgo a key aspect of their most effective abstractions, without offering a replacement.

If the term "computing" referred to the physical processes inside a computer, rather than to the abstraction, then a program in a programming language would not define a computation. One could define a computation only by describing the physical process. A computation is the same regardless of how it is executed. This consistency is, in fact, the essence of the abstraction. When considering CPS, it is arguable that we (the computer science community) have picked a rather inconvenient abstraction.

Moreover, the fact that the physical processes that implement computing take time is only one reason the abstraction is inconvenient. It would still be inconvenient if the physical process were infinitely fast. In order for computations to interact meaningfully with other physical processes, they must include time in the domain of discourse.

Time is a resource. Computation, as expressed in modern programming languages, obscures many resource-management problems. Memory is provided without bound by stacks and heaps. Power and energy consumption are (mostly) not the concern of a programmer. Even when resource-management problems are important to a particular application, there is no way for a programmer to talk about them within the semantics of a programming language.

Time is not like these other resources. First, barring metaphysical discourse, it is genuinely unbounded. To consider it a bounded resource, we would have to say that the available time per unit time is bounded, a tautology. Second, time is expended whether we use it or not. It cannot be conserved and saved for later. This is true, to a point, with, say, battery power, which is unquestionably a resource. Batteries leak, so their power cannot be conserved indefinitely, but designers rarely optimize a system to use as much battery power before it leaks away as they can. Yet that is what they do with time.

If time is indeed a resource, it is a rather unique one. Lumping together the problem of managing time with the problems of managing other more conventional resources inevitably leads to the wrong solutions. Conventional resource-management problems are optimization problems, not correctness problems. Using fewer resources is

always better than using more. Hence, there is no need to make energy consumption a semantic property of computing. Time, on the other hand, needs to be a semantic property.

Time is a nonfunctional property. What is the "function" of a program? In computation, it is a mapping from sequences of input bits to sequences of output bits (or an equivalent finite alphabet). The Turing-Church thesis defines "computable functions" as those that can be expressed by a terminating sequence of such bits-to-bits functions or mathematically by a finite composition of functions whose domain and co-domain are the set of sequences of bits.

In a CPS application, the function of a computation is defined by its effect on the physical world. This effect is no less a function than a mapping from bits to bits. It is a function in the intuitive sense of "what is the function of the system" and can be expressed as a function in the mathematical sense of a mapping from a domain to a co-domain.[15] But as a function, the domain and co-domain are not sequences of bits. Why do software designers insist on the wrong definition of "function"?

Designers of operating systems, Web servers, and communication protocols reactively view programs as a sequence of input/output events rather than as a mapping from bits to bits. This view needs to be elevated from the theoretical level to the application-programmer level and augmented with explicit temporal dynamics.

Real time is a quality-of-service (QoS) problem. Everybody, from architect to programmer to user, wants quality. Higher quality is always better than lower quality (at least under constant resource use). Indeed, in general-purpose computing, a key quality measure is execution time (or "performance"). But time in embedded systems plays a different role. Less time is not better than more time, as it is with performance. That less time is better than more time would imply that it is better for an automobile engine controller to fire the spark plugs earlier than later. Finishing early is not always a good thing and can lead to paradoxical behaviors where finishing early causes deadlines to be missed.[7] In an analysis that remains as valid today as it was when first spelled out by Stankovic[26] who lamented the resulting misconception that real-time computing "is equivalent to fast computing" or "is performance engineering." A CPS requires repeatable behavior much more than optimized performance.

Precision and variability in timing are QoS problems, but time itself is much more than a matter of QoS. If time is missing from the semantics of programs, then no amount of QoS will adequately address CPS timing properties.

Correctness

To solidify this discussion, I'll now define some terms based on the formal computational model known as the "tagged signal model."[15] A design is a description of a system; for example, a C program is a design, so is a C program together with a choice of microprocessor, peripherals, and operating system. The latter design (a C program combined with these design choices) is more detailed (less abstract) than the former.

More precisely, a *design* is a set of behaviors. A *behavior* is a valuation of observable variables, including all externally supplied inputs. These variables may themselves be functions; for example, in a very detailed design, each behavior may be a trace of electrical signals at the system's inputs and outputs. The *semantics* of a design is a set of behaviors.

In practice, a design is given in a design language that may be formal, informal, or some mixture of formal and informal. A design in a design language expresses the intent of the designer by defining the set of acceptable behaviors. Clearly, if the design language has precise (mathematical) semantics, then the set of behaviors is unambiguous. There could, of course, be errors in the expression, in which case the semantics will include behaviors not intended by the designer. For example, a function given in a pure functional programming language is a design. A designer can define a behavior to be a pair of inputs and outputs (arguments and results). The semantics of the program is the set of all possible behaviors that defines the function specified by the program. Alternatively, we could define a behavior to include timing information (when the input is provided and the output is produced); in this case, the semantics of the program includes all possible latencies (outputs can be produced arbitrarily later than the corresponding inputs), since nothing about the design language constrains timing.

A *correct execution* is any execution that is consistent with the semantics of the design. That is, given a certain set of inputs, a correct execution finds a behavior consistent with these inputs in the semantics. If the design language has loose or imprecise semantics, then "correct" executions may be unexpected. Conversely, if the design expresses every last detail of the implementation, down to printed circuit boards and wires, then a correct execution may, by definition, be any execution performed by said implementation. For the functional program just described, an execution is correct regardless of how long it takes to produce the output, because a program in a functional language says nothing about timing.

A *repeatable property* is a property of behaviors exhibited by every correct execution, given the same inputs; for example, the numerical value of the outputs of a pure functional program is repeatable. The timing of the production of the outputs is not. The timing can be made repeatable by giving more detail in the design by, for example, specifying a particular computer, compiler, and initial condition on caches and memory. The design has to get far less abstract to make timing repeatable.

A *predictable property* is a property of behaviors that can be determined in finite time through analysis of the design. That is, given only the information expressed in the design language, it needs to be possible to infer that the property is held by every behavior of a correct execution. For a particular functional program, the numerical value of the outputs may be predictable, but given an expressive-enough functional language, it will always be possible to write programs where these outputs are not predictable. If the language is Turing complete, then the numerical value of the outputs may be undecidable. In practice, even "finite time" is insufficient for a property to be predictable in practice. To be usefully predictable, properties must be inferred by a

programmer or program analysis tool in reasonable time.

Designs are generally abstractions of systems, omitting certain details. For example, even the most detailed design may not specify how behaviors change if the system is incinerated or crushed. However, an implementation of the design does have specific reactions to these events (albeit probably not predictable reactions). *Reliability* is the extent to which an implementation of a design delivers correct behaviors over time and under varying operating conditions. A system that tolerates more operating conditions or remains correct for a longer period of time is more reliable. Operating conditions include those in the environment (such as temperature, input values, timing of inputs, and humidity) but may also include those in the system itself (such as fault conditions like failures in communications and loss of power).

A *brittle* system is one in which small changes in the operating conditions or in the design yield incorrect behaviors. Conversely, a *robust* system remains correct with small changes in operating conditions or in design. Making these concepts mathematically precise is extremely difficult for most design languages, so engineers are often limited to intuitive and approximate assessments of these properties.

Requirements

Embedded systems have always been held to a higher reliability standard than general-purpose computing systems. Consumers do not expect their TVs to crash and reboot. They count on highly reliable cars in which computer controllers have dramatically improved both reliability and efficiency compared to electromechanical or manual controllers. In the transition to CPS, the expectation of reliability will only increase. Without improved reliability, CPS will not be deployed into such applications as traffic control, automotive safety, and health care in which human lives and property are potentially at risk.

The physical world is never entirely predictable. A CPS will not operate in controlled environments and must be robust to unexpected conditions and adaptable to subsystem failures. Engineers face an intrinsic tension between predictable performance and an unpredictable environment; designing reliable components makes it easier to assemble these components into reliable systems, but no component is perfectly reliable, and the physical environment will inevitably manage to foil reliability by presenting unexpected conditions. Given components that are reliable, how much can designers depend on that reliability when designing a system? How do they avoid brittle design?

The problem of designing reliable systems is not new in engineering. Two basic engineering tools are analysis and testing. Engineers analyze designs to predict behaviors under various operating conditions. For this analysis to work, the properties of interest must be predictable and yield to such analysis. Engineers also test systems under various operating conditions. Without repeatable properties, testing yields incoherent results.

Digital circuit designers have the luxury of working with a technology that delivers predictable and repeatable logical function and timing. This predictability and reliability holds despite the highly random underlying physics. Circuit designers have learned to harness intrinsically stochastic physical processes to deliver a degree of repeatability and predictability that is unprecedented in the history of human innovation. Software designers should be extremely reluctant to give up on the harnessing of stochastic physical processes.

The principle designers must follow is simple: Components at any level of abstraction should be made as predictable and repeatable as is technologically feasible. The next level of abstraction above these components must compensate for any remaining variability with robust design.

Some successful designs today follow this principle. It is (still) technically feasible to make predictable gates with repeatable behaviors that include both logical function and timing. Engineers design systems that count on these behaviors being repeatable. It is more difficult to make wireless links predictable and repeatable. Engineers compensate one level up, using robust coding schemes and adaptive protocols.

Is it technically feasible to make software systems that yield predictable and repeatable properties for a CPS? At the foundation of computer architecture and programming languages, software is essentially perfectly predictable and repeatable, if we consider only the properties expressed by the programming languages. Given an imperative language with no concurrency, well-defined semantics, and a correct compiler, designers can, with nearly 100% confidence, count on any computer with adequate memory to perform exactly what is specified in the program.

The problem of how to ensure reliable and predictable behavior arises when we scale up from simple programs to software systems, particularly to CPS. Even the simplest C program is not predictable and repeatable in the context of CPS applications because the design does not express properties that are essential to the system. It may execute perfectly, exactly matching its semantics (to the extent that C has semantics) yet still fail to deliver the properties needed by the system; it could, for example, miss timing deadlines. Since timing is not in the semantics of C, whether or not a program misses deadlines is irrelevant to determining whether it has executed correctly but is very relevant to determining whether the system has performed correctly. A component that is perfectly predictable and repeatable turns out not to be predictable and repeatable in the dimensions that matter. Such lack of predictability and repeatability is a failure of abstraction.

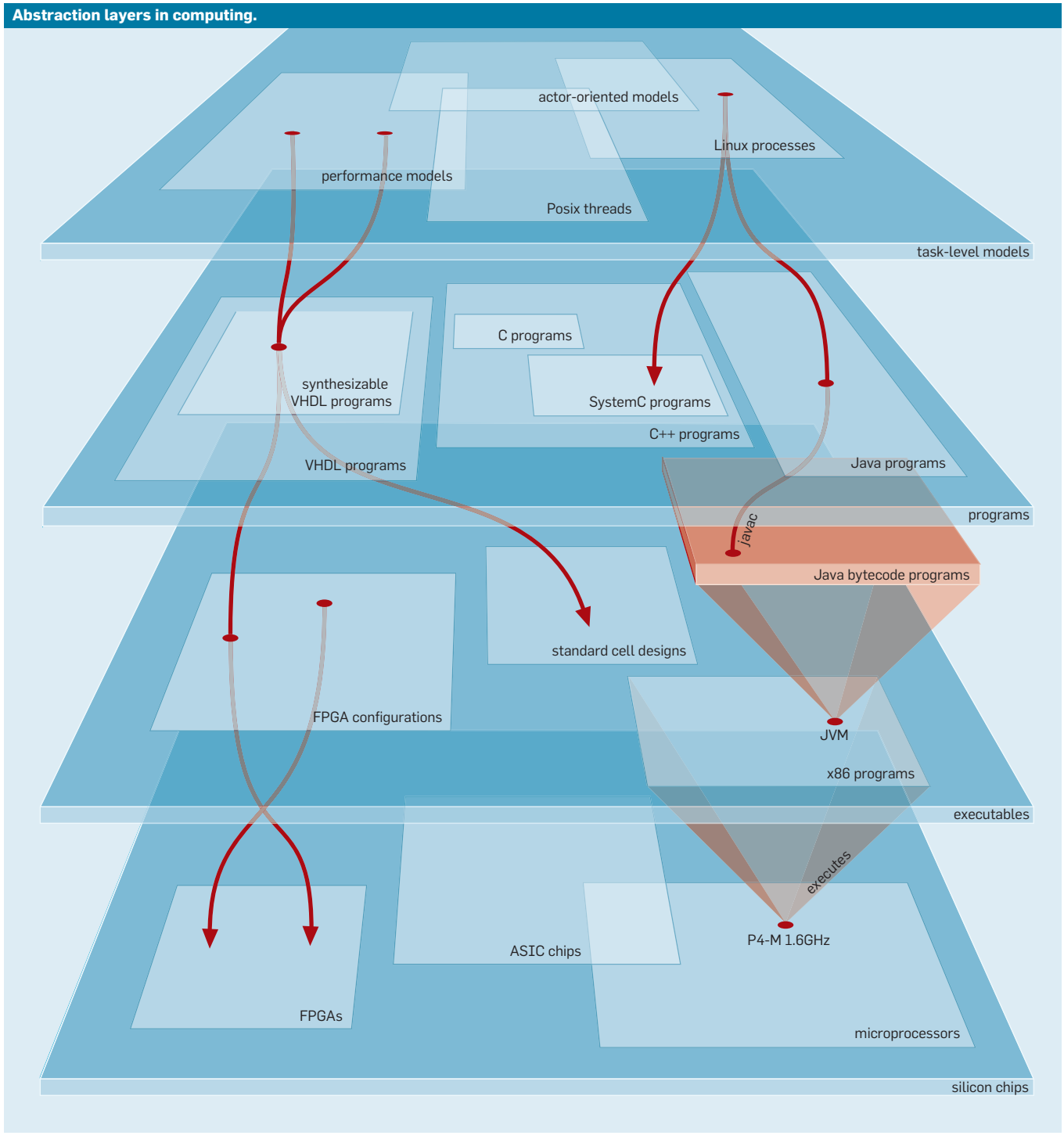
The problem of how to ensure predictable and repeatable behavior gets more difficult as software systems get more complex. If software designers step outside C and use operating system primitives to perform I/O or set up concurrent threads, they immediately move from essentially perfect predictability and repeatability to wildly nondeterministic behavior that must be carefully anticipated and reigned in by the software designer.[14] Semaphores, mutual exclusion locks, transactions, and priorities are some of the tools software designers have developed to attempt to compensate for the loss of predictability and repeatability.

But computer scientists must ask whether the loss of predictability and repeatability is necessary. No, it is not. If we find a way to deliver predictable

and repeatable timing, then we do not eliminate the core need to design robust systems but dramatically change the nature of the challenge. We must follow the principle of making systems predictable and repeatable, if technically feasible, and give up only when there is convincing evidence that delivering this result is not possible or cost-effective. There is no such evidence that delivering predictable and repeatable timing in software is not possible or cost effective. Moreover, we have an enormous asset: The substrate on which we build software systems (digital circuits) is essentially perfectly predictable and repeatable with respect to properties we most care about—timing and logical functionality.

Considering the enormous potential of CPS, I'll now further examine the failure of abstraction. The figure here schematically outlines some of the abstraction layers on which engineers depend when designing embedded systems. In the 3D Venn diagram, each box represents a set of designs. At the bottom is the set of all microprocessors; an element of this set (such as the Intel P4-M 1.6GHz) is a particular microprocessor design. Above that is the set of all x86 programs, each of which can run on that processor; this set is defined precisely (unlike the previous set of microprocessors, which is difficult to define) by the x86 instruction

Abstraction layers in computing.



set architecture (ISA). Any program coded in that instruction set is a member of the set; for example, a particular implementation of a Java virtual machine may be a member of the set. Associated with that member is another set—all JVM bytecode programs—each of which (typically) synthesized by a compiler from a Java program, which is a member of the set of all syntactically valid Java programs. Again, this set is defined precisely by Java syntax.

Each of these sets provides an abstraction layer intended to isolate a designer (the person or program that selects elements of the set) from the details below. Many of the best innovations in computing have come from careful and innovative construction and definition of these sets.

However, in the current state of embedded software, nearly every abstraction has failed. The ISA, meant to hide hardware implementation details from the software, has failed because ISA users care about timing properties ISA cannot express. The programming language, which hides details of ISA from the program logic, has failed because no widely used programming language expresses timing properties. Timing is an accident of implementation. A real-time operating system hides details of the program from their concurrent orchestration yet fails if the timing of the underlying platform is not repeatable or execution times cannot be determined. The network hides details of electrical or optical signaling from systems, but most standard networks provide no timing guarantees and fail to provide an appropriate abstraction. A system designer is stuck with a system design (not just implementation) in silicon and wires.

All embedded systems designers face this problem. For example, aircraft manufacturers must stockpile (in advance) the electronic parts needed for the entire production line of a particular aircraft model to ensure they don't have to recertify the software if the hardware changes. "Upgrading" a microprocessor in an engine control unit for a car requires thorough re-testing of the system.

Even "bug fixes" in the software or hardware can be extremely risky, since they can inadvertently change the system's overall timing behavior.

The design of an abstraction layer involves many choices, and computer scientists have uniformly chosen to hide timing properties from all higher abstractions. Wirth[30] says, "It is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time." However, in an embedded system, computations interact directly with the physical world, where time cannot be abstracted away.

Designers have traditionally covered these failures by finding worst-case execution time (WCET) bounds[29] and using real-time operating systems (RTOSs) with well-understood scheduling policies.[7] Despite recent improvements, these policies often require substantial margins for reliability, particularly as processor architectures develop increasingly elaborate techniques for dealing stochastically with deep pipelines, memory hierarchy, and parallelism.[11, 28]

Modern processor architectures render WCET virtually unknowable; even simple problems demand heroic efforts by the designer. In practice, reliable WCET numbers come with many caveats that are increasingly rare in software. Worse, any analysis that is done, no matter how tight the bounds, applies to only a specific program on a specific piece of hardware.

Any change in either hardware or software renders the analysis invalid. The processor ISA has failed to provide adequate abstraction. Worse, even perfectly tight WCET bounds for software components do not guarantee repeatability. The so-called "Richard's anomalies," explained nicely by Buttazzo,[7] show that under popular earliest-deadline first (EDF) scheduling policies, the fact that all tasks finish early might cause consequential deadlines to be missed that would not have been missed if the tasks had finished at the WCET bound. Designers must be very careful to analyze their scheduling strategies under worst-case and best-case execution times, along with everything in between.

Timing behavior in RTOSs is coarse and increasingly uncontrollable as the complexity of the system increases (such as by adding inter-process communication). Locks, priority inversion, interrupts, and similar concurrency issues break the formalisms, forcing designers to rely on bench testing that is often incapable of identifying subtle timing bugs. Worse, these techniques generally produce brittle systems in which small changes cause big failures.

While there are no absolute guarantees in life, or in computing, we should not blithely discard achievable predictability and repeatability. Synchronous digital hardware—the most basic technology on which computers are built—reliably delivers astonishingly precise timing behavior. However, software abstractions discard several orders of magnitude of precision. Compare the nanosecond-scale precision with which hardware raises an interrupt request to the millisecond-level precision with which software threads respond. Computer science doesn't have to do it this way.

Solutions

The timing problems I raise here pervade computing abstractions from top to bottom. As a consequence, most specialties within the field have work to do. I suggest a few directions, all drawn from existing contributions, suggesting that the vision I've outlined, though radical, is indeed achievable. We do not need to restart computer science from scratch.

Computer architecture. The ISA of a processor provides an abstraction of computing hardware for the benefit of software designers. The value of this abstraction is enormous, including that generations of CPUs that implement the same ISA can have different performance numbers without compromising compatibility with existing software. Today's ISAs hide most temporal properties of the underlying hardware. Perhaps the time is right to augment the ISA abstraction with carefully selected timing properties, so the compatibility extends to time-sensitive systems. This is the objective of a new generation of "precision timed" machines.[9]

Achieving timing precision is easy if system designers are willing to forgo performance; the engineering challenge is to deliver both precision and performance. For example, although cache memories may introduce unacceptable timing variability, cost-effec-

tive system design cannot do without memory hierarchy. The challenge is to provide memory hierarchy with repeatable timing. Similar challenges apply to pipelining, bus architectures, and I/O mechanisms.

Programming languages. Programming languages provide an abstraction layer above the ISA. If the ISA is to expose selected temporal properties and programmers wish to exploit the exposed properties, then one approach would be to reflect these properties in the languages.

There is a long and somewhat checkered history of attempts by language developers to insert timing features into programming languages. For example, Ada can express a delay operation but not timing constraints. Real-Time Java augments the Java model with ad-hoc features that reduce the variability of timing. The synchronous languages[5] (such as Esterel, Lustre, and Signal) lack explicit timing constructs but, in light of their predictable and repeatable approach to concurrency, can yield more predictable and repeatable timing than most alternatives. They are limited only by the underlying platform. In the 1970s, Modula-2 gave control over scheduling of co-routines, making it possible, albeit laboriously, for programmers to exercise coarse control over timing. Like the synchronous languages, timing properties of programs developed with Modula-2 are not explicit in the program. Real-time Euclid, on the other hand, expresses process periods and absolute start times.

Rather than create new languages, an alternative is to annotate programs written in conventional languages. For example, Lee[16] gave a taxonomy of timing properties that must be expressible in such annotations. TimeC[17] introduces extensions to specify timing requirements based on events, with the objective of controlling code generation in compilers to exploit instruction-level pipelining. Domain-specific languages with temporal semantics have taken hold in some. For example, Simulink, from The MathWorks, provides a graphical syntax and language for timed systems that can be compiled into embedded real-time code for control systems. LabVIEW, from National Instruments, which is widely used in instrumentation systems, recently added timed extensions. Another example from the 1970s, PEARL,[22] was also aimed at control systems and could specify absolute and relative start times, deadlines, and periods.

In an embedded system, computations interact directly with the physical world, where time cannot be abstracted away.

However, all these programming environments and languages remain outside the mainstream of software engineering, are not well integrated into software engineering processes and tools, and have not benefited from many innovations in programming languages.

Software components. Software engineering innovations (such as data abstraction, object-orientation, and component libraries) have made it much easier to design large complex software systems. Today's most successful component technologies—class libraries and utility functions—do not export even the most rudimentary temporal properties in their APIs. Although a knowledgeable programmer may be savvy enough to use a hash table over a linked list when random access is required, the API for these data structures expresses nothing about access times. Component technologies with temporal properties are required, providing an attractive alternative to real-time programming languages. An early example from the mid-1980s, Larch,[4] gave a task-level specification language that integrated functional descriptions with timing constraints. Other examples function at the level of coordination language rather than specification language. A coordination language executes at runtime; a specification language does not.

For example, Broy[6] focused on timed concurrent components communicating via timed streams. Zhao et al.[31] developed an actor-based coordination language for distributed real-time systems based on discrete-event systems semantics. New coordination languages, where the components are given using established programming languages (such as Java and C++), may be more likely to gain acceptance than new programming languages that replace established languages. When coordination languages are given rigorous timed semantics, designs function more like models than like programs.

Many challenges remain in developing coordination languages with

timed semantics. Naïve abstractions of time (such as the discrete-time models commonly used to analyze control and signal-processing systems) do not reflect the true behavior of software and networks.[23] The concept of "logical execution time"[10] offers a more promising abstraction but ultimately relies on being able to achieve worst-case execution times for software components. This top-down solution depends on a corresponding bottom-up solution.

Formal methods. Formal methods use mathematical models to infer and prove system properties. Formal methods that handle temporal dynamics are less prevalent than those that handle sequences of state changes, but there is good work on which to draw. For example, in interface theories,[8] software components export temporal interfaces, and behavioral-type systems validate the composition of components and infer interfaces for compositions of components; for specific interface theories of this type, see Kopetz and Suri[13] and Thiele et al.[27]

Various temporal logics support reasoning about the timing properties of systems.[3] Temporal logics mostly deal with "eventually" and "always" properties to reason about safety and liveness, and various extensions support metric time.[1,21] A few process algebras also support reasoning about time.[19, 24] The most accepted formalism for the specification of real-time requirements is timed automata and its variations.[2]

Another approach widely used in instrumentation systems uses static analysis of programs coupled with models of the underlying hardware.[29] Despite gaining traction in industry, it suffers from fundamental limitations, with brittleness the most important. Even small changes in either the hardware or the software invalidate the analysis. A less-important limitation, though worth noting, is that the use of Turing-complete programming languages and models leads to undecidability. In other words, not all programs can be analyzed.

All these techniques enable some form of formal verification. However, properties that are not formally specified cannot be formally verified. Thus, for example, the timing behavior of software that is not expressed in the software must be separately specified,

Designers must be very careful to analyze their scheduling strategies under worst-case and best-case execution times, along with everything in between.

and the connections between specifications and between specification and implementations become tenuous. Moreover, despite considerable progress in automated abstraction, scalability to real-world systems remains a challenging hurdle. Although offering a wealth of elegant results, the effect of most of these formal techniques on engineering practice has been small (though not zero). In general-purpose computing, type systems are formal methods that have had enormous effect by enabling compilers to catch many programming errors. What is needed is *time systems* with the power of type systems.

Operating systems. Scheduling is a key service of any operating system, and scheduling of real-time tasks is a venerable, established area of inquiry in software design. Classic techniques (such as rate-monotonic scheduling and EDF) are well studied and have many elaborations. With a few exceptions,[10, 12] the field of operating system development has seen less emphasis on repeatability over optimization. Repeatability is not highly valued in general-purpose applications. Consider this challenge: To get repeatable real-time behavior, a CPS designer may use the notion of logical execution time (LET)[10] for the time-sensitive portions of a system and best-effort execution for the less-time-sensitive portions. The best-effort portions typically have no deadlines, so EDF gives them lowest priority. However, the correct optimization is to execute the best-effort portions as early as possible, subject to the constraint that the LET portions match their timing specifications. Even though the LET portions have deadlines, they should not necessarily be given higher priority during program execution than the best-effort portions.

Designers of embedded systems deliberately avoid mixing time-sensitive operations with best-effort operations. Every cellphone in use has at least two CPUs, one for the difficult real-time tasks of speech coding and radio functions, the other for the user interface, database, email, and networking functionality. The situation is more complicated in cars and manufacturing systems, where distinct CPUs tend to be used for myriad distinct features. The design is this way, not because there are

not enough cycles in the CPUs to combine the tasks, but because software designers lack reliable technology for mixing distinct types of tasks. Focusing on repeatability of timing behavior could lead to such a mixing technology; work on deferrable/sporadic servers[18] may provide a promising point of departure.

Networking. In the context of general-purpose networks, timing behavior is viewed as a QoS problem. Considerable activity in the mid-1980s to mid-1990s led to many ideas for addressing QoS concerns, few of which were deployed with any long-lasting benefit. Today, designers of time-sensitive applications on general-purpose networks (such as voice over IP) struggle with inadequate control over network behavior.

Meanwhile, in embedded systems, specialized networks (such as FlexRay and the time-triggered architecture[12]) have emerged to provide timing as a correctness property rather than as a QoS property. A flurry of recent activity has led to a number of innovations (such as time synchronization, IEEE 1588), synchronous Ethernet, and time-triggered Ethernet). At least one of them—synchronous Ethernet—is encroaching on general-purpose networking, driven by demand for convergence of telephony and video services with the Internet, as well as by the potential for real-time interactive games. However, introducing timing into networks as a semantic property rather than as a QoS problem inevitably leads to an explosion of new time-sensitive applications, helping realize the CPS vision.

Conclusion

Realizing the potential of CPS requires first rethinking the core abstractions of computing. Incremental improvements will continue to help, but effective orchestration of software and physical processes requires semantic models that reflect properties of interest in both.

I've focused on making temporal dynamics explicit in computing abstractions so timing properties become correctness criteria rather than a QoS measure. The timing of programs and networks should be as repeatable and predictable as is technologically feasible at reasonable cost. Repeatability and predictability will not eliminate timing variability and hence not eliminate the need for adaptive techniques and validation methods that work with bounds on timing. But they do eliminate spurious sources of timing variability, enabling precise and repeatable timing when needed. The result will be computing and networking technologies that enable vastly more sophisticated CPS applications.

Acknowledgments

Special thanks to Tom Henzinger, Insup Lee, Al Mok, Sanjit Seshia, Jack Stankovic, Lothar Thiele, Reinhard Wilhelm, Moshe Vardi, and the anonymous reviewers for their helpful comments and suggestions. C

References

1. Abadi, M. and Lamport, L. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept. 1994), 1543–1571.
2. Alur, R. and Dill, D.L. A theory of timed automata. *Theoretical Computer Science 126*, 2 (Apr. 1994), 183–235.
3. Alur, R. and Henzinger, T. Logics and models of real time: A survey. In *Real-Time: Theory in Practice: Proceedings of the REX Workshop, Vol. 600 LNCS*, J.W. De Bakker, C. Huizing, W.P. De Roever, and G. Rozenberg, Eds. (Mook, The Netherlands, June 3–7). Springer, Berlin/Heidelberg 1991, 74–106.
4. Barbacci, M.R. and Wing, J.M. *Specifying Functional and Timing Behavior for Real-Time Applications, Technical Report ESD-TR-86-208.* Carnegie Mellon University, Pittsburgh, PA, Dec. 1986.
5. Benveniste, A. and Berry, G. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE 79*, 9 (Sept. 1991), 1270–1282.
6. Broy, M. Refinement of time. *Theoretical Computer Science 253*, 1 (Feb. 2001), 3–26.
7. Buttazzo, G.C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Second Edition.* Springer, Berlin/Heidelberg, 2005.
8. deAlfaro, L. and Henzinger, T.A. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software, Vol. LNCS 2211*. Springer, Berlin/Heidelberg, 2001, 148–165.
9. Edwards, S.A. and Lee, E.A. The case for the precision timed (PRET) machine. In *Proceedings of the Design Automation Conference* (San Diego, CA, June 4–8). ACM Press, New York. 2007, 264–265.
10. Henzinger, T.A., Horowitz, B., and Kirsch, C.M. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software, Vol. LNCS 2211*. Springer, Berlin/Heidelberg, 2001, 166–184.
11. Kirner, R. and Puschner, P. Obstacles in worst-case execution time analysis. In *Proceedings of the Symposium on Object-Oriented Real-Time Distributed Computing* (Orlando, FL, May 5–7). IEEE Computer Society Press. New York, 2008, 333–339.
12. Kopetz, H. and Bauer, G. The time-triggered architecture. *Proceedings of the IEEE 91*, 1 (Jan. 2003), 112–126.
13. Kopetz, H. and Suri, N. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Hakodate, Hokkaido, Japan, May 14–16). IEEE Computer Society Press, 2003, 51–60.
14. Lee, E.A. The problem with threads. *Computer 39*, 5 (May 2006), 33–42.
15. Lee, E.A and Sangiovanni-Vincentelli, A. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems 17*, 12 (Dec. 1998), 1217–1229.
16. Lee, I., Davidson, S., and Wolfe, V. *Motivating Time as a First-Class Entity, Technical Report MS-CIS-87-54.* Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, Aug. (Revised Oct.) 1987.
17. Leung, A., Palem, K.V., and Pnueli, A. *TimeC: A Time-Constraint Language for ILP Processor Compilation, Technical Report TR1998-764.* New York University, New York, 1998.
18. Liu, J.W.S. *Real-Time Systems.* Prentice-Hall, Upper Saddle River, NJ, 2000.
19. Liu, X. and Lee, E.A. *CPO Semantics of Timed Interactive Actor Networks, Technical Report EECS-2006-67.* University of California, Berkeley, May 18, 2006.
20. Maler, O., Manna, Z., and Pnueli, A. In *Real-Time: Theory in Practice: Proceedings of the REX Workshop, Vol. 600 LNCS*, J.W. De Bakker, C. Huizing, W.P. De Roever, and G. Rozenberg, Eds. (Mook, The Netherlands, June 3–7). Springer, Berlin/Heidelberg, 1991, 447–484.
21. Manna, Z and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, Berlin, 1992.
22. Martin, T. Real-time programming language PEARL: Concept and characteristics. In *Proceedings of the Computer Software and Applications Conference*, IEEE Press, 1978, 301–306.
23. Nghiem, T., Pappas, G.J., Girard, A., and Alur, R. Time-triggered implementations of dynamic controllers. In *Proceedings of 6th ACM & IEEE Conference on Embedded Software* (Seoul, Korea, Oct. 23–25). ACM Press, New York, 2006, 2–11.
24. Reed, G.M. and Roscoe, A.W. A timed model for communicating sequential processes. *Theoretical Computer Science 58*, 1–3 (June 1988), 249–261.
25. Stankovic, J.A., Lee, I., Mok, A., and Rajkumar, R. Opportunities and obligations for physical computing systems. *Computer 38*, 11 (Nov. 2005), 23–31.
26. Stankovic, J.A. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer 21*, 10 (Oct. 1998), 10–19.
27. Thiele, L., Wandeler, E., and Stoimenov, N. Real-time interfaces for composing real-time systems. In *Proceedings of Sixth ACM & IEEE Conference on Embedded Software* (Seoul, Korea, Oct. 23–25). ACM Press, New York, 2006, 34–43.
28. Thiele, L. and Wilhelm, R. Design for timing predictability. *Real-Time Systems 28*, 2–3 (Nov. 2004), 157–177.
29. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenstr, P. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems 7*, 3 (Apr. 2008), 1–53.
30. Wirth, N. Toward a discipline of real-time programming. *Commun. ACM 20*, 8 (Aug. 1977), 577–583.
31. Zhao, Y., Lee, E.A., and Liu, J. A programming model for time-synchronized distributed real-time systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium* (Bellevue, WA, Apr. 3–6). IEEE Computer Society Press, New York. 2007, 1–10.

This work is supported in part by the Center for Hybrid and Embedded Software Systems at the University of California, Berkeley, which receives support from the U.S. National Science Foundation, Army Research Office, Air Force Office of Scientific Research, Air Force Research Lab, State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, and Toyota. For an extended version go to www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-30.html.

Edward A. Lee (eal@eecs.berkeley.edu) is the Robert S. Pepper Distinguished Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley.

DOI:10.1145/1506409.1506427

The convergence of CS and biology will serve both disciplines, providing each with greater power and relevance.

BY CORRADO PRIAMI

Algorithmic Systems Biology

THROUGHOUT THE HISTORY of computer science, leading researchers—including Turing, von Neumann, and Minsky—have looked to nature. This inspiration has often led to extraordinary results, some of which acknowledged biology even in their names: cellular automata, neural networks, and genetic algorithms, for example.

Computing and biology have been converging ever more closely for the past two decades, but with a vision of computing as a resource for biology. The resulting field of bioinformatics addresses structural aspects of biology, and it has produced databases, pattern manipulation and comparison methods, search tools, and data-mining techniques.[47,48] Bioinformatics' most notable and successful application so far has been the Human Genome Project, which was made possible by the selection of the correct abstraction for representing DNA (a language with a four-character alphabet).[48] But things are now proceeding in the reverse direction as well. Biology is experiencing a heightening of interest in system dynamics by interpreting living organisms as information manipulators.[30] It is thus moving toward "systems biology."[31] There is no general agreement on systems biology's definition, but whatever we select must embrace at least four characterizing concepts. Systems biology is a transition:

- From qualitative biology toward a quantitative science;
- From reductionism to system-level understanding of biological phenomena;
- From structural and static descriptions to functional and dynamic properties; and
- From descriptive biology to mechanistic/causal biology.

These features highlight the fact that causality between events, the temporal ordering of interactions and the spatial distribution of components are becoming essential to addressing biological questions at the system level. This development poses new challenges to describing the step-by-step mechanistic components of phenotypical phenomena, which bioinformatics does not address.[9]

One of the philosophical foundations of systems biology is mathematical modeling, which specifies and tests hypotheses about systems;[7] it is also a key aspect of computational biology because it deals with the solution of systems of equations (models) through computer programs.[37] Solution of systems of equations is sometimes termed "simulation." By whatever name, the main concept to be exploited involves instead algorithms and the (programming) languages used to specify them. We can then recover temporal, spatial, and causal information on the modeled systems by using well-established computing techniques that deal with program analysis, composition, and verification; integrated software-development environments; and debugging tools as well as computational complexity and algorithm animation. The convergence between computing and systems biology on a peer-to-peer basis is then a valuable opportunity that can

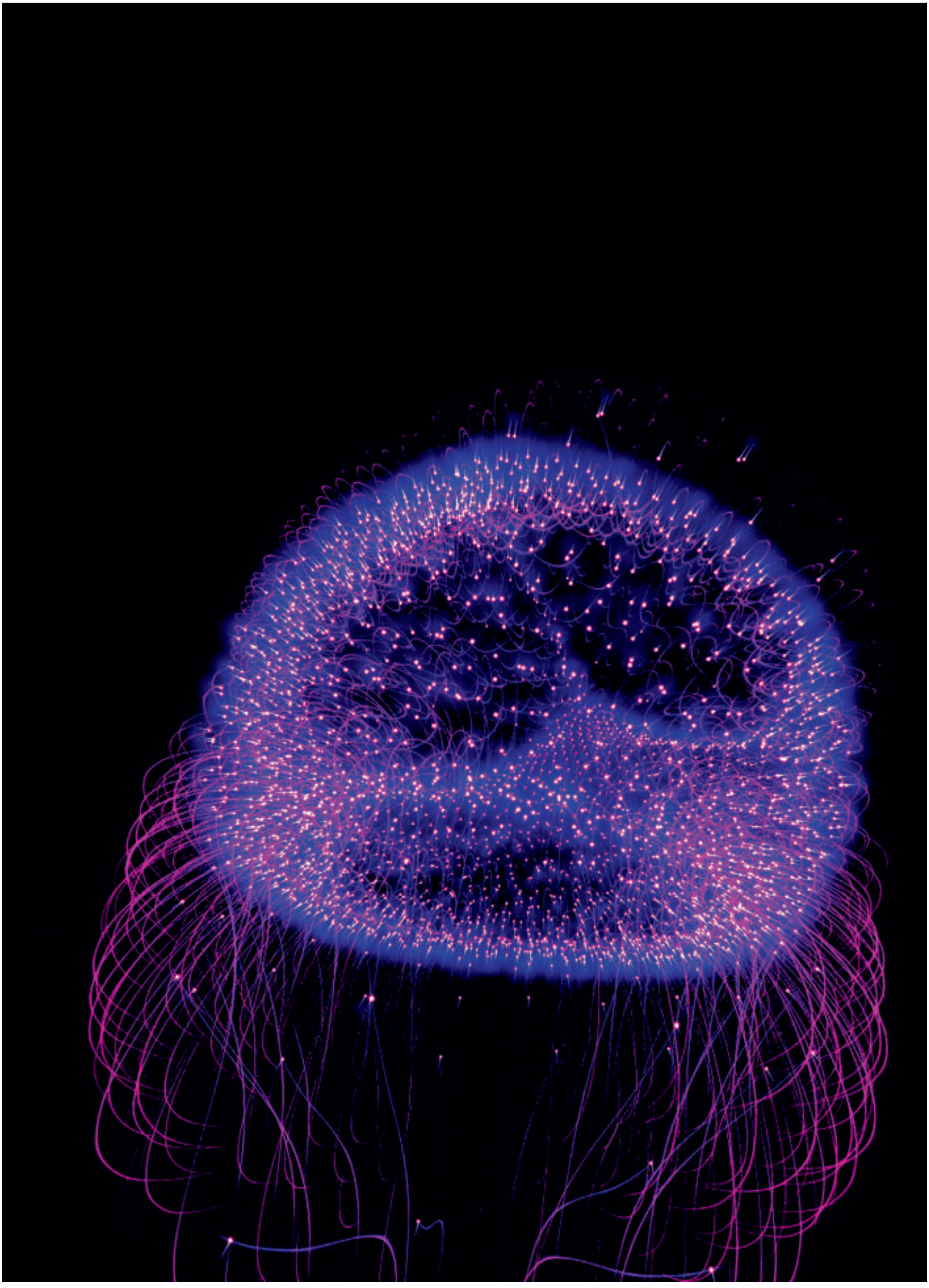


ILLUSTRATION BY ROBERT HODGIN

fuel the discovery of solutions to many of the current challenges in both fields, thereby moving toward an algorithmic view of systems biology.

The main distinction between algorithmic systems biology and other techniques used to model biological systems stems from the intrinsic difference between algorithms (operational descriptions) and equations (denotational descriptions). Equations specify dynamic processes by abstracting the steps performed by the executor, thus hiding from the user the causal, spatial, and temporal relationships between those steps. Equations describe the changing of variables' values when a system moves from one state to another, while algorithms highlight *why* and *how* that system transition occurs. We could simplify the difference by stating that we move from the pictures described by equations to the film described by algorithms.

Algorithms precisely describe the behavior of systems with discrete state spaces, while equations describe an average behavior of systems with continuous state spaces. However, it must be noted that hybrid approaches exist; they manipulate discrete state spaces annotated with continuous variables through algorithms.[2, 15]

It is well known in computer science that input-output relationships are not suitable for characterizing the behavior of concurrent systems, where many threads of execution are simultaneously active (in biological systems, millions of interactions may be involved). Concurrency theory was developed as a formal framework in which to model and analyze parallel, distributed, and mobile systems, and this led to the definition of specific programming primitives and algorithms. Equations, by contrast, are sequential tools that attempt to model a system whose behavior is completely determined by input-output relations. The sequential assumption of equations also impacts the notion of causality that coincides with the temporal ordering of events. In a parallel context, causality is instead a function of concurrency[14] and may not coincide with the temporal ordering of the observed events. Therefore relying on a sequential modeling style to describe an inherently concurrent system immediately makes the modeler lose the connection with causality.

The full involvement of computer science in systems biology can be an arena in which to distinguish between computing and mathematics, thereby clarifying a discussion that has been going on for 40 years.[21, 33] Algorithms and the coupling of executions/executors are key to that differentiation.

Algorithms force modelers/biologists to think about the mechanisms governing the behavior of the system in question. Therefore they are both a conceptual tool that helps to elucidate fundamental biological principles and a practical tool for expressing and favoring computational thinking.[53] Similar ideas have been recently expressed in Ciocchetta et al.[12]

Algorithms are quantitative when the mechanism for selection of the next step is based on probabilistic/temporal distributions associated with either the rules or the components of the system being modeled. Because the dynamics of biological systems are mainly driven by quantities such as concentrations, temperatures, and gradients, we must clearly focus on quantitative algorithms and languages.

Algorithms can help in coherently extracting general biological principles that underlie the enormous amount of data produced by high-throughput technologies. Algorithms can also organize data in a clear and compact way, thus producing knowledge from information (data). This point actually aligns with the idea of Nobel laureate Sydney Brenner that biology needs a theory able to highlight causality and abstract data into knowledge so as to elucidate the architecture of biological complexity.

Algorithms need an associated syntax and semantics in order to specify their intended meaning so that an executor can precisely and unambiguously perform the steps needed to implement them. In this way, we are entering the realm of programming languages from both a theoretical and practical perspective.

The use of programming languages to model biological systems is an emerging field that enhances current modeling capabilities (richness of aspects that can be described as well as the easiness, composability, and reusability of models).[42] The underlying metaphor is one that represents biological entities as programs being executed simultaneously and that represents the interactions of two entities by the exchange of messages between the programs.[46] The biological entities involved in a biological process and the corresponding programs in the abstract model are in a 1:1 correspondence, thus avoiding the need to deal directly with the combinatorial explosion of variables needed in the mathematical approach.

Figure 1: Algorithms enable a transformation from "pictures" to "films." The current practice in biological systems entails modeling the variation of measures through equations, with no causal explanation given (upper part of the figure). But algorithms describe the steps from one picture to the next in a causal continuum of the actions that make the measures change, thus providing a dynamic view of the system in question.

This metaphor explicitly refers to concurrency. Indeed, concurrency is endemic in nature, and we see this in examples ranging from atoms to molecules in living organisms to the organisms themselves to populations to astronomy. If we are going to reengineer artificial systems to match the efficiency, resilience, adaptability, and robustness of natural systems, then concurrency must be a core design principle that, at the end of the day, will simplify the entire design and implementation process. Concurrency, therefore, must not be considered as just a tool to improve the performance of sequential-programming languages and architectures, which is the standard practice in most actual cases.

Some programming languages—those that address concurrency as a core primitive issue and that aim at modeling biological systems—are in fact emerging from the field of process calculi.[5, 15] These concurrent programming languages are very promising for establishing a link between artificial concurrent programming and natural phenomena, thus contributing to the exposure of computer science to experimental natural sciences. Further, concurrent programming languages are suitable candidates for easily and efficiently expressing the mechanistic rules that propel algorithmic systems biology. The suitability of these languages is reinforced by their clean and formal definition, which supports both the verification of properties and the analysis of systems and provides no engineering surprises, as could happen with classical thread and lock mechanisms.[52]

A recent paper by Nobel laureate Paul Nurse maintains that a better understanding of living organisms requires "both the development of the appropriate languages to describe information processing in biological systems and the generation of more effective methods to translate biochemical descriptions into the functioning of the logic circuits that underpin biological phenomena."[38] This description perfectly reflects the need for a deeper involvement of computer science in biology and the need of an algorithmic description of life based on a suitable language that makes analyses easier. Nurse's statement implicitly assumes that the modeling techniques adopted so far are not adequate to address the new challenges raised by systems biology.

Design principles of large software systems can help in developing an algorithmic discipline not only for systems biology but also for synthetic biology—a new area of research that aims at building, or synthesizing, new biological systems and functions by exploiting new insights from science and engineering.

Finally, it is important to note that process calculi are not the only theoretical basis of algorithmic systems biology. Petri nets, logic, rewriting systems, and membrane computing are other relevant examples of formal methods applied to systems biology (for a collection of tutorials see Bernardo et al[6]). Other approaches that are more closely related to software-design principles are the adaptation of UML to biological issues (see www.biouml.org) and statecharts.[28] Finally, cellular automata[27] need to be considered as well, with their game of life.

The Role of Computing

According to Denning,[18] the field of computing addresses information processes, both artificial and natural, by manipulating different layers of abstraction at the same time and structuring their automation on a machine.[53] These abilities are relevant in the convergence of computer science and biology in assuring that the correct modeling abstraction for biological systems be found that is not created solely by mimicking nature. A beautiful example of this statement is that airplanes do not flap their wings, though they fly.[50]

Augmenting the range of applicative domains taken from other fields is the main strategy for making computer science grow as a discipline, for improving the core themes developed so far, and for making it more accessible to a broader community.[32] Therefore adding an algorithmic component to systems biology is an especially valuable opportunity, as this new approach covers, in a unique challenge, four core practices of computer science: programming, systems thinking, modeling, and innovation.[17] Algorithmic systems biology is a bona fide case of innovation fostered by computer science, as it uses novel ideas for modeling and analyzing experiments. Moreover, the biotech and pharmaceuticals industries could adopt algorithmic systems biology in order to streamline their organizations and internal processes with the aim of improving productivity.[52]

To have an impact on the scientific community and to truly foster innovation, algorithmic systems biology must provide conceptual and software tools

that address real biological problems. Hence the techniques and prototypes developed and tested on proof-of-concept examples must scale smoothly to real-life case studies. Scalability is not a new issue in computing; it was first raised several decades ago when computers started to become connected over dispersed geographical regions and the first high-performance architectures were emerging.[39] Algorithmic systems biology can build on the large set of successfully defined and novel techniques that subsequently were developed—particularly in the areas of programming languages, operating systems, and software-development environments—to address the scalable specification and implementation of large distributed systems.

Consider the exploitation, at user level, of the Internet. The dynamics (evolution and use) of the Internet have no centralized point of control and are based both on the interaction between nodes and on the unpredictable birth and death of new nodes—characteristics similar to the simultaneously active threads of interactions in living systems. Yet although biological processes share many similarities with the dynamics of large computer networks, they still have some unique features. These include self-reproduction of components (dating back to von Neumann's self-reproducing automata in computing and to Rosen's systems, closed under efficient causality, in biology), auto-adaptation to different environments, and self-repair. Therefore it seems natural to check whether the programming and analysis techniques developed for computer networks and their formal theories could shed light on biology when suitably adapted.

Such innovation should be facilitated in the life sciences community by presenting computers as high-throughput tools for quantitatively analyzing information processes and systems—tools that can be made greatly customized through software to work with specific processes or systems. In other words, software may be used to plan and control information experiments to serve a myriad of purposes.

The topic of simulation in particular needs some consideration. Simulation has evolved since the early days of computing into a more quantitative algorithmic discipline: the rules of interaction between components are used to build programs, as opposed to abstracting overall behavior through equations. The execution of algorithmic simulations relies on deep computing theories, while mathematical simulations are solved with the support of computer programs (where computing is just a service).[22] Execution of algorithms therefore exhibits emergent behavior produced at system level, through the set of local interactions between components, without the need to specify that behavior from the beginning. This property is crucial to the predictive power of the simulation approach, especially for biological applications. The complex interactions of species, the sensitivity of their interactions (expressed through stochastic parameters), and the localization of the components in a three-dimensional hierarchical space make it impossible to understand the dynamic evolution of a biological system without a computational execution of the models. The algorithms that are executed on top of stochastic engines and governed by the quantities described here are fundamental to discovering new organismic behavior and thus to creating new biological hypotheses.

Algorithmic systems biology completely adopts the main assets of our computing discipline: hierarchical, systems, and algorithmic thinking in modeling, programming and innovating.

Algorithmic systems biology can also be easily integrated with bioinformatics. An example that would benefit from such integration is the modeling of the immune system, because the dynamics of an immune response involve a genomic resolution scale in addition to the dimensions of time and space. Inserting genomic sequences of viruses into models is quite easy for an algorithmic modeling approach, but it is extremely difficult in a classic mathematical model,[45] which suffers from generalization because a population of heterogeneous agents is usually abstracted into a single continuous variable.[8] Deepening our understanding of the immune system through computing models is fundamental to properly attacking infectious illnesses such as malaria or HIV as well as autoimmune diseases that include rheumatoid arthritis and type I diabetes. Also, computer science can exploit such models to further propel research on artificial immune systems in the field of security.[23]

Design principles of large software systems can help in developing an algo-

rithmic discipline not only for systems biology but also for synthetic biology—a new area of biological research that aims at building, or synthesizing, new biological systems and functions by exploiting new insights from science and engineering. An algorithmic approach can help propel this field by providing an in-silico library of biological components that can be used to derive models of large systems; such models could be ready for simulation and analysis just by composing the available modules.[16]

The notion of a library of (biological) components, equipped with attributes governing their interaction capabilities and automatically exploited by the implementation of the language describing systems dynamics, substantially contributes to overcoming the misleading concept of pathways that fills biological papers, where a pathway is posited as an almost-sequential chain of interactions. The theory of concurrency, however, maintains that neglecting the context of interactions (all the other possible routes of the system) produces an incomplete and untrustworthy understanding of the system's dynamics. Metaphorically, it is not possible to understand the capacity of the traffic organization of a city by looking at single routes from one point to another, or to fully appreciate a goal in a team sport by looking at the movements of a single player.

At a different level of abstraction, the study of pathways is a reductionist approach that does not take pathway interactions (crosstalk) into account and does not help in unraveling emergent network behavior. The management of hierarchies of interconnected specifications, so typical of computer science, is fundamental for interpreting what systems behavior means, depending on the context and the properties of interest. It could be easy to move to biological networks by considering the biological entities as a collection of interacting processes and by studying the behavior of the network through the conceptual tools of concurrency theory.

Note also that a model repository, representing the dynamics of biological processes in a compact and mechanistic manner, would be extremely valuable in heightening the understanding of biological data and the basic principles governing life. Such a repository would favor predictions, allow for the optimal design of further experiments, and consequently stimulate the movement from data collection to knowledge production.

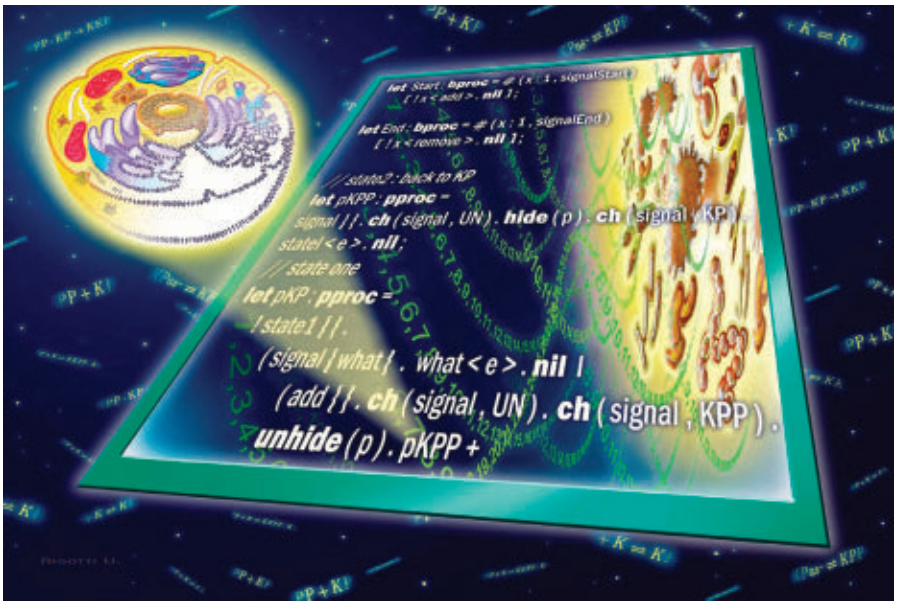


Figure 2: The biological systems observed through the window showing the life sciences (green rectangle) can be closely and mechanistically modeled through the use of algorithms (written on the glass of the window) that add causal, spatial, and temporal dimensions to classical biological descriptions. Moreover, algorithms can concisely represent the large quantities of data produced by high-throughput experiments (the river of numbers originating from biological elements within the window). Equations, currently considered the stars of modeling, are more abstract and hence more distant from living matter. The goal of algorithmic systems biology is to "reach for the moon" through a complete mechanistic model of living systems. (The lighted hemisphere in the picture represents a cell under a digitalization process.)

Algorithmic systems biology raises novel issues in computing by stepping away from the qualitative descriptions typical of programming languages toward a new quantitative computing. Thus computing can fully become an experimental science, as advocated by Denning,[20] that is suitable to supporting systems biology. Core computing fields would themselves benefit from a quantitative approach; a measure of the level of satisfaction from Web service contracts, for example, or the quality of services in telecommunication networks could enhance our current software-development techniques. Another example is robotics, where a myriad of sensors must be synchronized according to quantitative values. Quantitative computing would also foster the move toward a simulation-based science that is needed to address the increasingly larger dimension and complexity of scientific questions.

It will easily become impossible to have the whole system we design available for testing (examples are the new Boeing and Airbus aircraft) and hence we need to find alternatives for studying and validating the system's behavior. Simulation of formal specifications is one possibility. Indeed, the programming languages used to model biological systems implement stochastic runtime supports that help in addressing extremely relevant questions in biology such as "How does order emerge from disorder?"[44] The answers could provide us with completely new ways of organizing robust and self-adapting networks both natural and technological. Further, the discrete-state nature of algorithmic descriptions makes them suitable for implementing the stochastic simulation algorithm by Gillespie[25] or its variants. This approach, originally developed for biochemical simulations, is also suitable for quantitatively simulating systems from other domains; in fact, there are cases in which it can be much faster than classical event-driven simulation.[34]

Algorithmic systems biology completely adopts the main assets of our computing discipline: hierarchical, systems, and algorithmic thinking in modeling, programming and innovating. Moreover, because breakthrough results are sometimes the outcome of

processes that do not perfectly adhere to the scientific method based on experiment and observations, creativity can play a crucial role in opening minds and propelling visions of new findings in the future. In fact, we can further exploit algorithmic descriptions of biology for the synthesis (in silico) of completely new organisms by using our conceptual tools in an imaginative way that is similar to the engineering of novel solutions and applications via software in computer science. This approach would parallel that of another emerging field—synthetic biology—which aims to create unnatural systems assembled from natural components to study their behavior. Synthesis, in other words, is a fundamental process that allows us to understand phenomena that cannot be easily captured by analysis and modeling. For instance, the synthesis of a minimal cell would help in understanding the fundamental principles of self-replicating systems and evolution, which are the core elements of life.[10, 24] Once again, computer science is a perfect vehicle for such inquiry, in which analysis and synthesis are always interwoven. Hence its past experience can substantially help in addressing the key issues of systems and synthetic biology.

Challenges and Future Directions

The main challenges inherent in building algorithmic models for the system-level understanding of biological processes include the relationship between low-level local interactions and emergent high-level global behavior; the partial knowledge of the systems under investigation; the multilevel and multiscale representations in time, space, and size; the causal relations between interactions; and the context-awareness of the inner components. Therefore the modeling formalisms that are candidates for propelling algorithmic systems biology should be complementary to and interoperable with mathematical modeling, address parallelism and complexity, be algorithmic and quantitative, express causality, and be interaction-driven, composable, scalable, and modular.

Composability—the ability to characterize a system starting from the descriptions of its subsystems and a set of rules for assembling them—is fundamental to addressing the complexity of the applicative domain and at the same time to exploiting the benefits of parallel architectures such as many-multicore processors. Composability can either be shallow (that is, syntactic) or deep (semantic).[13] Algorithmic systems biology needs both of these aspects of composability: models of biological systems must be built by shallow composition of building blocks taken from a library, and the specification of the overall system's behavior must be obtained by deep composition of the representation of the building blocks' behavior.

A relevant example relates to interactions, which can be studied on the molecular-machinery level (at one extreme) and on the population level (at the other). Metagenomics—the analysis of complex ecosystems as metaorganisms or complex biological networks—is an exciting and challenging field in which algorithms could help explain fundamental phenomena that are still not completely understood; one such phenomenon is horizontal gene transfer between bacteria (whereby bacteria exchange pieces of genome within the same generation to improve their adaptation to the environment, as in developing resistance to antibiotics). The success of these investigations is strictly tied to the identification of the right level of abstraction within the hierarchies of interactions (from molecules to organisms). Because the comprehension of how life organizes itself into cells, organisms, and communities is a major challenge that systems biology strives to understand,[40] and because computer science is continuously shifting between various coherent views of the same artificial system, depending on the properties of interest, its capabilities could be crucial in addressing such issues in natural systems.

Another example is rhythmic behavior, which is so common in biological systems that understanding it is crucial to unraveling the dynamics of life.[26] Rhythms have been a key point of interest in mathematical and computational biology since their earliest days[51]—a century of studies identified feedback processes (both positive/forward and negative/backward loops) and cooperativity as main sources of unstable behavior. These general control structures have a strong similarity to the primitives of concurrent programming languages used to specify the flow of control—for instance, the dichotomy of the cooperation and competition of processes to access resources; the infinite behavior of drivers of resources in operating systems; and conditional guarded commands to choose the next step to be performed. Once again, the full theory developed to cope with concurrency in artificial systems perfectly couples with algorithmic descriptions of biological systems, yielding a new reference framework in which computer science is a novel foundation for studying and understanding cellular rhythms.

Multiscale integration (in space, time, and size) is a major issue in current systems biology[3] as well. The very essence of the multiple levels of abstraction that govern computer science—enabling it to address phenomena that span several orders of magnitude (from one clock cycle [nanoseconds] to a whole computations [hours])[21]—can help unravel and master the complexity of genome-wide modeling of biological systems. Thus the dynamic relationship between the parts and the whole of a system that seems to be the essence of systems biology is also a keystone for managing artificial (computing) systems. Such a relationship was even used to define computer science.[36]

Another relevant aspect of systems biology is the sensitivity of a network's behavior to the quantitative parameters that govern its dynamics[41, 51]—for instance, the concentration of species in a system or their affinity for interaction affects the speed of the reactions, thereby affecting the system's overall behavior. Current developments such as variance or uncertainty output analysis usually consider a biological system as a black box that implements a function from inputs to outputs, assuming the system is deterministic.[35] But as discussed earlier, I/O relationships are not the best way, or even a correct way, of defining the semantics of concurrent programs; different runs with the same inputs may generate different outcomes because of the relative speed of subcomponents. Given that biological systems are massively concurrent—not deterministic—a new algorithmic language-based modeling approach can certainly create new avenues for the sensitivity analysis of networks. That is, simulation-based science can

turn sensitivity analysis of highly parallel systems into an observation-driven analysis, based on model-checking and verification techniques developed over the last 30 years for concurrent systems. The new findings could in turn benefit computer science itself.

Algorithmic systems biology will be innovative and successful if the life-sciences community actually uses the available conceptual and computational tools for modeling, simulation, and analysis. To ease this task, computing tools must hide as many formal details as possible from users, and here the growing and important area of software visualization can play a critical role. Visual metaphors of algorithm animations will help biologists understand how systems evolve, even while the scientists remain bound to their classical "picture" representations. Such pictures, however, may more profitably be mapped into "films."

A final remark pertains to the comparison of different systems. Equivalences are a main tool in computer science for verifying computing systems; they can be used, for instance, to ensure that an implementation is in agreement with a specification. They abstract as much as possible from syntactic descriptions and instead focus on specifications' and implementations' semantics. So far, biology has focused on syntactic relationships between genes, genomes, and proteins, but an entirely new avenue of research is the investigation of the semantic equivalences of biological entities' interactions in complex networks. This approach could lead to new visions of systems and reinforce computer science's ability to enhance systems biology.

Algorithmic systems biology can contribute to the future both of life sciences and natural sciences through interconnecting models and experiments.

Impact

The integration of computer science and systems biology into algorithmic systems biology is a win-win strategy that will affect both disciplines, scientifically and technologically.

The scientific impact of accomplishing our vision, aided by feedback from the increased understanding of basic biological principles, will be in the definition of new quantitative theoretical frameworks. These frameworks can then help us address the increasing concurrency and complexity—observed in asynchronous, heterogeneous, and decentralized (natural/artificial) systems—in a verifiable, modular, incremental, and composable manner. Further, the definition of novel mechanisms for quantitative coordination and orchestration will produce new conceptual frameworks able to cope with the growing paradigm of distributing the logic of application between local software and global services. The definition of new schemas to store data related to the dynamics of systems and the new query languages needed to retrieve and examine such records will create novel perspectives. They, in turn, will allow the building of data centers that provide added value to globally available services.

Another major scientific impact will be the definition of a new philosophical foundation of systems biology that is algorithmic in nature and allows scientists to raise new questions that are out of range for the current conceptual and computational supports. An example is the interpretation of pathways (which do not exist per se in nature) as a reductionist approach for understanding the behavior of networks (collections of interwoven pathways interacting and working simultaneously) at the system level.

The technological impact of merging computer science and systems biology will be the design and implementation of artificial biology laboratories capable of performing many more experiments than what is currently feasible in real labs—and at lower cost (in terms both of human and financial resources) and in less time. These labs will allow biologists to design, execute, and analyze experiments to generate new hypotheses and develop novel high-throughput tools, resulting in advances in experimental design, documentation, and interpretation as well as a deeper integration between "wet" (lab-based) and "dry" research. Moreover, the artificial biology laboratories will be a main vehicle for moving from single-gene diseases to multifactorial diseases, which account for more than 90% of the illnesses affecting our society.

A deeper look at the causes of multifactorial diseases can positively influence their diagnosis and management. But health is not the only practical application of algorithmic systems biology. Comprehension of the basic mech-

anisms of life, coupled with engineered environments for synthetic biological design, can lead us toward the use of ad hoc bacteria to repair environmental damages as well as to produce energy.

Another major technological impact will be computer scientists' ability to properly address the challenges posed by the hardware revolution—increasingly stressing parallelism in place of speed of processors—through new integrated programming environments amenable to concurrency and complexity.

Conclusion

Quantitative algorithmic descriptions of biological processes add causal, spatial, and temporal dimensions to molecular machinery's behavior that is usually hidden in the equations. Algorithmic systems biology allows us to take a step forward in our understanding of life by transforming collections of pictures (cartoons) into spectacular films (the mechanistic dynamics of life). In fact, the languages and algorithms emerging from quantitative computing can be instrumental not only to systems biology but also to the scientific understanding of interactions in general.

Unraveling the basic mechanisms adopted by living organisms for manipulating information goes to the heart of computer science: computability. Life underwent billions of years of tests and was optimized during this very long time; we can learn new computational paradigms from it that will enhance our field. The same arguments apply to hardware architectures as well. Starting from the basics, we can use these new computational paradigms to strengthen resource management and hence operating systems, to develop primitives to instruct highly parallel systems and hence (concurrent) programming languages, and to develop software environments that ensure higher quality and better properties than current software applications.

Algorithmic systems biology can contribute to the future both of life sciences and natural sciences through interconnecting models and experiments. New conceptual and computational tools, integrated in a user-friendly environment, can be employed by life scientists to predict the behavior of multilevel and multiscale biological systems, as well as of other kinds of systems, in a modular, composable, scalable, and executable manner.

Algorithmic systems biology can also contribute to the future of computer science by developing a new generation of operating systems and programming languages. They will enable advanced simulation-based research, within a quantitative framework that connects in-silico replicas and actual systems, and enabled by biologically inspired tools.

Acknowledgments

The author thanks the CoSBi team for the numerous inspiring discussions.

References

1. Alon, U. *An Introduction to Systems Biology: Design Principles of Biological Circuits.* Chapman and Hall, 2006.
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. The algorithmic analysis of hybrid systems. *Theoretical Computer Science 138*, 1 (1995), 3–34.
3. Auffray, C. and Nottale, L. Scale relativity theory and integrative systems biology: Founding principles and scale laws. *Progress in Biophysics and Molecular Biology 97* (2008), 79–114.
4. Benner, S.A. and Sismour, A.M. Synthetic biology. *Nature Reviews Genetics 6* (2005), 533–544.
5. Bergstra, J.A., Ponse, A., and Smolka, S.A. *Handbook of Process Algebras.* Elsevier, 2001.
6. Bernardo, M., Degano, P., and Zavattaro, G. *Formal Methods for Computational Systems Biology.* LNCS 5016, Springer, 2008.
7. Boogerd, F. et al. *Systems Biology: Philosophical Foundations.* Elsevier, 2007.
8. Breckling, B. Individual-based modelling: Potentials and limitations. *Scientific World Journal 2* (April 19, 2002), 1044–1062.
9. Cassman, M., Arkin, A., Doyle, F., Katagiri, F., Lauffenburg, D., and Stokes, C. *International Research and Development in Systems Biology.* WTEC Panel on Systems Biology final report (Oct. 2005).
10. Chiarugi, D., Degano, P., and Marangoni, R. A computational approach to the functional screening of genomes. *PLoS Comput Biol 3*, 9 (Sept. 3, 2007), 1801–1806.
11. Ciocchetta, F. and Hillston, J. Process algebras in systems biology. In *Formal Methods for Computational Systems Biology, LNCS 5016*. Springer, 2008, 313–365.
12. Cohen, J. The crucial role of CS in systems and synthetic biology. *Commun. ACM 51*, 5 (May 2008), 15–18.
13. de Alfaro, L., Henzinger, T.A., and Jhala, R. Compositional methods for probabilistic systems. In *CONCUR01, LNCS 2154* (2001).
14. Degano, P. and Priami, C. Non-interleaving semantics of mobile processes. *Theoretical Computer Science 216* 1–2 (1999), 237–270.
15. Dematté, L., Priami, C., and Romanel, A. The BlenX language: A tutorial. In *Formal Methods for Computational Systems Biology, LNCS 5016*. Springer, 2008. 313-365.
16. Dematté, L., Priami, C., Romanel, A. The Beta Workbench: A tool to study the dynamics of biological systems. *Briefings in Bioinformatics*, 2008.
17. Denning, P.J. Great principles of computing. *Commun. ACM 46*, 11 (Nov. 2003), 15–20.
18. Denning, P.J. Is computer science science? *Commun. ACM 48*, 4 (Apr. 2005), 27–31.
19. Denning, P.J. Recentering computer science. *Commun. ACM 48*, 11 (Nov. 2005), 15–19.
20. Denning, P.J. Computing is a natural science. *Commun. ACM 50*, 5 (July 2007),13–18.
21. Dijstra, E.W. Programming as a discipline of mathematical nature. *American Mathematical Monthly 81* (1974), 608–612.
22. Fisher, J. and Henzinger, T. Executable cell biology. *Nature Biotechnology 25* (2007), 1239–1249.
23. Forrest, S. and Beauchemin, C. *Imm. Reviews 216* (2007), 176–197.
24. Foster, A.C., Church, G.M. Towards synthesis of a minimal cell. *Molecular Systems Biology* (2006).
25. Gillespie, D.T. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry 81* (1977), 2340–2361.
26. Goldbeter, A. Computational approaches to cellular rhythms. *Nature 420* (2002), 238–245.
27. Gutowitz, H. Introduction (to cellular automata). *Physica D 45*, 1990.
28. Harel, D. Statecharts in the making: A personal account. In *Proceedings of the 3rd ACM SIGPLAN History of Programming Languages Conference*, 2007.
29. Hendler, J., Shadbolt, N., Hall, W., Berners-Lee, T., and Weitzner, D. Web science: An interdisciplinary approach to understanding the Web. *Comm. ACM 51*, 7 (July 2008), 60–69.
30. Hood, L., Galas, D. The digital code of DNA. *Nature 421* (2003), 444–448.
31. Kitano, H. Systems biology: A brief overview. *Science 295* (2002), 1662–1664.
32. Klawe, M. and Shneiderman, B. Crisis and opportunity in computer science. *Commun. ACM 48,* 11 (Nov. 2005).
33. Knuth, D. Computer science and its relation to mathematics. *American Mathematical Monthly 81* (1974), 323–343.
34. Kuwahara, H. and Mura, I. An efficient and exact stochastic simulation method to analyze rare events in biological systems. *Journal of Chemical Physics 129*, 2008.
35. Ludtke, N., Panzeri, S., Brown, M., Broomhead, D.S., Knowles, J., Montemurro M.A., and Kell, D.B. Information-theoretic sensitivity analysis: A general method for credit assignment in complex networks. *J. R. Soc. Interface 5* (2008), 223–235.
36. Minsky, M. ACM Turing Lecture: Form and content in computer science. *Journal of the ACM 17*, 2 (1970), 197–215.
37. Nature insight: Computational biology. *Nature 420* (2002), 206–251.
38. Nurse, P. Life, logic and information. *Nature 454* (2008), 424–426.
39. Nussbaum, D. and Agarwal, A. Scalability of parallel machines. *Commun. ACM 34*, 3 (Mar. 1991), 57–61.
40. O'Malley, M. and Dupré, J. Fundamental issues in systems biology. *BioEssays, 27* (2005), 1270–1276.
41. Palsson, B.O. *Systems Biology: Properties of Reconstructed Networks.* Cambridge University Press, 2006.
42. Priami, C. and Quaglia, P. Modeling the dynamics of biosystems. *Briefings in Bioinformatics 5* (2004), 259–269.
43. Priami, C., Regev, A., Shapiro, E., and Silvermann, W. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters 80* (2001), 25–31.
44. Rao, C.V., Wolf, D.M., and Arkin, A.P. Control, exploitation and tolerance of intracellular noise. *Nature 420* (2002), 231–237.
45. Rapin, N., Kesmir, C., Frankild, S., Nielsen, M., Lundegaard, C., Brunak, S., and Lund, O. Modeling the human immune system by combining bioinformatics and systems biology approaches. *Journal Biol. Phys. 32* (2006), 335–353.
46. Regev, A. and Shapiro, E. Cells as computation. *Nature 419* (2002), 343.
47. Roos, D. *Bioinformatics*—Trying to swim in a sea of data. *Science 291* (2001), 260–1261.
48. Searls, D. The language of genes. *Nature 420* (2002), 211–217.
49. Spengler, S.J. Bioinformatics in the information age. *Science 287* (2000), 1221–1223.
50. Teuscher, C. Biologically uninspired computer science. *Commun. ACM 49*, 11 (Nov. 2006), 27–29.
51. Volterra, V. Fluctuations in the abundance of species considered mathematically. *Nature 118* (1926), 558–560.
52. Welch, P.H. and Barnes, F.R.M. Communicating mobile processes: Introducing occam-pi. In *CSP25, LNCS 3525*. Springer, 2005, 175–210.
53. Wing, J. Computational thinking. *Commun. ACM 49*, 3 Mar. 2006), 33–35.

Corrado Priami (Priami@cosbi.eu) is president and CEO of the Microsoft Research-University of Trento Centre for Computational and Systems Biology and professor of Computer Science at the Department of Engineering and Information Sciences of the University of Trento.

research highlights

DOI:10.1145/1506409.1506428

Technical Perspective
A Chilly Sense of Security

By Ross Anderson

THE FOLLOWING PAPER by Alex Halderman et al. will change the way people write and test security software.

Many systems rely on keeping a master key secret. Sometimes this involves custom hardware, such as a smartcard, and sometimes it relies on an implicit hardware property, such as the assumption that a computer's RAM loses state when it is powered off. And software writers tend to assume that hardware works in the intuitively obvious ways.

But technological progress can undermine old assumptions.

Years ago, Sergei Skorobogatov showed that memory cells used in microcontrollers could retain their contents for many minutes at low temperatures; an attacker could freeze a chip to stop its keys evaporating while he depackaged it and probed out the contents.

That was long thought to be an arcane result of relevance only to engineers designing crypto boxes for banks and governments. But, as this paper illustrates, progress has made memory remanence (as it is known) relevant to the "ordinary" software business, too. Modern memory chips, when powered down, will retain their contents for seconds even at room temperature, and for minutes if they are cooled to the temperatures of a Canadian winter.

The upshot is that your laptop encryption software is no longer secure.

The key used to protect disk files is typically kept in RAM, so a locked laptop can be unlocked by cooling it, interrupting the power, rebooting with a new operating system kernel, and reading out the key.

Even if a few bits of the key have decayed, common implementations of both DES and AES keep redundant representations of the key in memory to improve performance; these not only provide error correction but enable keys to be found quickly.

This neat piece of work emphasizes once more the need for engineers who build security applications to take a holistic view of the world.

For their pièce de résistance, the authors show how to break BitLocker, the disk encryption utility in Microsoft Vista, and the culmination of the 10-year, multibillion-dollar "Trusted Computing" research program. BitLocker was believed to be strong because the master keys are kept in the TPM chip on the motherboard while the machine is powered down. Hundreds of millions of PCs now have TPM chips; your PC cost a few dollars more as a result. But did it make your PC more secure? It turns out that keys remain in memory so long as the machine is powered up; and worse, they are loaded to memory when the machine is powered on, before the user ever has to enter a password. In either case, the memory remanence attack can suck them up just fine. The upshot is that you're less secure than before. An old-fashioned disk encryption utility can at least protect your data when your machine is powered down. Adding "hardware security" has undermined even that.

This neat piece of work emphasizes once more the need for engineers who build security applications to take a holistic view of the world.

Software alone is not enough; you need to understand the hardware, and the people too.

Ross Anderson (Ross.Anderson@cl.cam.ac.uk) is a professor of security engineering at the University of Cambridge, England.

DOI:10.1145/1506409.1506429

Lest We Remember: Cold-Boot Attacks on Encryption Keys

By J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten

Abstract

Contrary to widespread assumption, dynamic RAM (DRAM), the main memory in most modern computers, retains its contents for several seconds after power is lost, even at room temperature and even if removed from a motherboard. Although DRAM becomes less reliable when it is not refreshed, it is not immediately erased, and its contents persist sufficiently for malicious (or forensic) acquisition of usable full-system memory images. We show that this phenomenon limits the ability of an operating system to protect cryptographic key material from an attacker with physical access to a machine. It poses a particular threat to laptop users who rely on disk encryption: we demonstrate that it could be used to compromise several popular disk encryption products without the need for any special devices or materials. We experimentally characterize the extent and predictability of memory retention and report that remanence times can be increased dramatically with simple cooling techniques. We offer new algorithms for finding cryptographic keys in memory images and for correcting errors caused by bit decay. Though we discuss several strategies for mitigating these risks, we know of no simple remedy that would eliminate them.

1. INTRODUCTION

Most security practitioners have assumed that a computer's memory is erased almost immediately when it loses power, or that whatever data remains is difficult to retrieve without specialized equipment. We show that these assumptions are incorrect. Dynamic RAM (DRAM), the hardware used as the main memory of most modern computers, loses its contents gradually over a period of seconds, even at normal operating temperatures and even if the chips are removed from the motherboard. This phenomenon is called *memory remanence*. Data will persist for minutes or even hours if the chips are kept at low temperatures, and residual data can be recovered using simple, nondestructive techniques that require only momentary physical access to the machine.

We present a suite of attacks that exploit DRAM remanence to recover cryptographic keys held in memory. They pose a particular threat to laptop users who rely on disk encryption products. An adversary who steals a laptop while an encrypted disk is mounted could employ our attacks to access the contents, even if the computer is screen-locked or suspended when it is stolen.

On-the-fly disk encryption software operates between the file system and the storage driver, encrypting disk blocks as they are written and decrypting them as they are read. The encryption key is typically protected with a password typed by the user at login. The key needs to be kept available so that programs can access the disk; most implementations store it in RAM until the disk is unmounted.

The standard argument for disk encryption's security goes like this: As long as the computer is screen-locked when it is stolen, the thief will not be able to access the disk through the operating system; if the thief reboots or cuts power to bypass the screen lock, memory will be erased and the key will be lost, rendering the disk inaccessible. Yet, as we show, memory is not always erased when the computer loses power. An attacker can exploit this to learn the encryption key and decrypt the disk. We demonstrate this risk by defeating several popular disk encryption systems, including BitLocker, TrueCrypt, and FileVault, and we expect many similar products are also vulnerable.

Our attacks come in three variants of increasing resistance to countermeasures. The simplest is to reboot the machine and launch a custom kernel with a small memory footprint that gives the adversary access to the residual memory. A more advanced attack is to briefly cut power to the machine, then restore power and boot a custom kernel; this deprives the operating system of any opportunity to scrub memory before shutting down. An even stronger attack is to cut the power, transplant the DRAM modules to a second PC prepared by the attacker, and use it to extract their state. This attack additionally deprives the original BIOS and PC hardware of any chance to clear the memory on boot.

If the attacker is forced to cut power to the memory for too long, the data will become corrupted. We examine two methods for reducing corruption and for correcting errors in recovered encryption keys. The first is to cool the memory chips prior to cutting power, which dramatically prolongs data retention times. The second is to apply algorithms we have developed for correcting errors in private and symmetric keys. These techniques can be used alone or in combination.

While our principal focus is disk encryption, any sensitive data present in memory when an attacker gains physical access to the system could be subject to attack. For example, we found that Mac OS X leaves the user's login password in memory, where we were able to recover it. SSL-enabled Web

The full version of this paper was published in *Proceedings of the 17th USENIX Security Symposium*, August 2008, USENIX Association. The full paper, video demonstrations, and source code are available at http://citp.princeton.edu/memory/.

servers are vulnerable, since they normally keep in memory private keys needed to establish SSL sessions. DRM systems may also face potential compromise; they sometimes rely on software to prevent users from accessing keys stored in memory, but attacks like the ones we have developed could be used to bypass these controls.

It may be difficult to prevent all the attacks that we describe even with significant changes to the way encryption products are designed and used, but in practice there are a number of safeguards that can provide partial resistance. We suggest a variety of mitigation strategies ranging from methods that average users can employ today to long-term software and hardware changes. However, each remedy has limitations and trade-offs, and we conclude that there is no simple fix for DRAM remanence vulnerabilities.

Certain segments of the computer security and hardware communities have been conscious of DRAM remanence for some time, but strikingly little about it has been published. As a result, many who design, deploy, or rely on secure systems are unaware of these phenomena or the ease with which they can be exploited. To our knowledge, ours is the first comprehensive study of their security consequences.

2. CHARACTERIZING REMANENCE

A DRAM cell is essentially a capacitor that encodes a single bit when it is charged or discharged.[10] Over time, charge leaks out, and eventually the cell will lose its state, or, more precisely, it will decay to its *ground state*, either zero or one depending on how the cell is wired. To forestall this decay, each cell must be *refreshed*, meaning that the capacitor must be recharged to hold its value—this is what makes DRAM "dynamic." Manufacturers specify a maximum *refresh interval*—the time allowed before a cell is recharged—that is typically on the order of a few milliseconds. These times are chosen conservatively to ensure extremely high reliability for normal computer operations where even infrequent bit errors can cause problems, but, in practice, a failure to refresh any individual DRAM cell within this time has only a tiny probability of actually destroying the cell's contents.

To characterize DRAM decay, we performed experiments on a selection of recent computers, listed in Figure 1. We filled representative memory regions with a pseudorandom test pattern, and read back the data after suspending refreshes for varying periods of time by cutting power to the machine. We measured the error rate for each sample as the number of bit errors (the Hamming distance from the pattern we had written) divided by the total number of bits. Fully decayed memory would have an error rate of approximately 50%, since half the bits would match by chance.

Figure 1: Test Systems. We experimented with six systems (designated A–F) that encompass a range of recent DRAM architectures and circuit densities.

	Density	Type	System	Year
A	128MB	SDRAM	Dell Dimension 4100	1999
B	512MB	DDR	Toshiba Portégé R100	2001
C	256MB	DDR	Dell Inspiron 5100	2003
D	512MB	DDR2	IBM Thinkpad T43p	2006
E	512MB	DDR2	IBM Thinkpad x60	2007
F	512MB	DDR2	Lenovo 3000 N100	2007

2.1. Decay at operating temperature

Our first tests measured the decay rate of each machine's memory under normal operating temperature, which ranged from 25.5°C to 44.1°C. We found that the decay curves from different machines had similar shapes, with an initial period of slow decay, followed by an intermediate period of rapid decay, and then a final period of slow decay, as shown in Figure 2.

The dimensions of the decay curves varied considerably between machines, with the fastest exhibiting complete data loss in approximately 2.5 s and the slowest taking over a minute. Newer machines tended to exhibit a shorter time to total decay, possibly because newer chips have higher density circuits with smaller cells that hold less charge, but even the shortest times were long enough to enable some of our attacks. While some attacks will become more difficult if this trend continues, manufacturers may attempt to *increase* retention times to improve reliability or lower power consumption.

We observed that the DRAMs decayed in highly nonuniform patterns. While these varied from chip to chip, they were very stable across trials. The most prominent pattern is a gradual decay to the ground state as charge leaks out of the memory cells. In the decay illustrated in Figure 3, blocks of cells alternate between a ground state of zero and a ground state of one, resulting in the horizontal bars. The fainter vertical bands in the figure are due to manufacturing variations that cause cells in some parts of the chip to leak charge slightly faster than those in others.

Figure 2: Measuring decay. We measured memory decay after various intervals without power. The memories were running at normal operating temperature, without any special cooling. Curves for machines A and C would be off the scale to the right, with rapid decay at around 30 and 15 s, respectively.

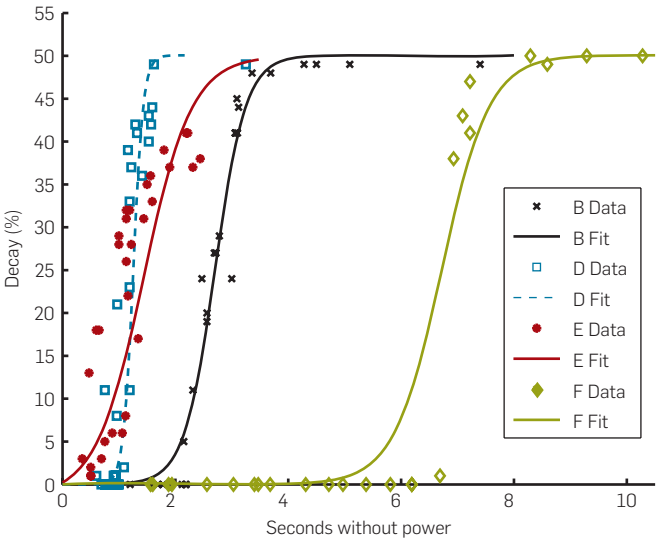
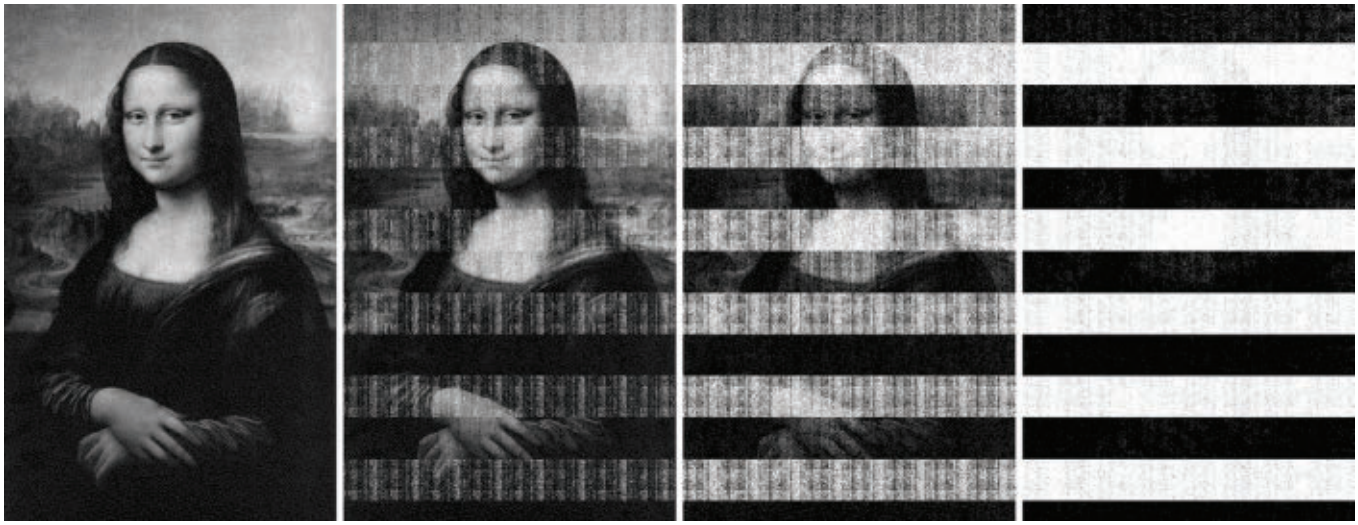


Figure 3: Visualizing memory decay. We loaded a bitmap image into memory on test machine A, then cut power for varying intervals. After 5 s (*left*), the image is nearly indistinguishable from the original; it gradually becomes more degraded, as shown after 30, 60 s, and 5 min. The chips remained close to room temperature. Even after this longest trial, traces of the original remain. The decay shows prominent patterns caused by regions with alternating ground states (*horizontal bars*) and by physical variations in the chip (*fainter vertical bands*).



2.2. Decay at reduced temperature

Colder temperatures are known to increase data retention times. We performed another series of tests to measure these effects. On machines A–D, we loaded a test pattern into memory, and, with the computer running, cooled the memory module to approximately –50°C. We then cut power to the machine and maintained this temperature until power and refresh were restored. As expected, we observed significantly slower rates of decay under these reduced temperatures (see Figure 4). On all of our test systems, the decay was slow enough that an attacker who cut power for 1 min would recover at least 99.9% of bits correctly.

We were able to obtain even longer retention times by cooling the chips with liquid nitrogen. After submerging the memory modules from machine A in liquid nitrogen for 60 min, we measured only 14,000 bit errors within a 1MB test region (0.13% decay). This suggests that data might be recoverable for hours or days with sufficient cooling.

Figure 4: Colder temperatures slow decay. We measured memory errors for machines A–D after intervals without power, first at normal operating temperatures (no cooling) and then at a reduced temperature of –50°C. Decay occurred much more slowly under the colder conditions.

		Average Bit Errors	
	Seconds without Power	No Cooling (%)	–50°C (%)
A	60	41	[no errors]
	300	50	0.000095
B	360	50	[no errors]
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

3. TOOLS AND ATTACKS

Extracting residual memory contents requires no special equipment. When the system is powered on, the memory controller immediately starts refreshing the DRAM, reading and rewriting each bit value. At this point, the values are fixed, decay halts, and programs running on the system can read any residual data using normal memory-access instructions.

One challenge is that booting the system will necessarily overwrite some portions of memory. While we observed in our tests that the BIOS typically overwrote only a small fraction of memory, loading a full operating system would be very destructive. Our solution is to use tiny special-purpose programs that, when booted from either a warm or cold reset state, copy the memory contents to some external medium with minimal disruption to the original state.

Most modern PCs support network booting via Intel's Preboot Execution Environment (PXE), which provides rudimentary start-up and network services. We implemented a tiny (9KB) standalone application that can be booted directly via PXE and extracts the contents of RAM to another machine on the network. In a typical attack, a laptop connected to the target machine via an Ethernet crossover cable would run a client application for receiving the data. This tool takes around 30 s to copy 1GB of RAM.

Some recent computers, including Intel-based Macintosh systems, implement the Extensible Firmware Interface (EFI) instead of a PC BIOS. We implemented a second memory extractor as an EFI netboot application. Alternatively, most PCs can boot from an external USB device such as a USB hard drive or flash device. We created a third implementation in the form of a 10KB plug-in for the SYSLINUX

bootloader. It can be booted from an external USB device or a regular hard disk.

An attacker could use tools like these in a number of ways, depending on his level of access to the system and the countermeasures employed by hardware and software. The simplest attack is to reboot the machine and configure the BIOS to boot the memory extraction tool. A warm boot, invoked with the operating system's restart procedure, will normally ensure that refresh is not interrupted and the memory has no chance to decay, though software will have an opportunity to wipe sensitive data. A cold boot, initiated using the system's restart switch or by briefly removing power, may result in a small amount of decay, depending on the memory's retention time, but denies software any chance to scrub memory before shutting down.

Even if an attacker cannot force a target system to boot memory extraction tools, or if the target employs countermeasures that erase memory contents during boot, an attacker with sufficient physical access can transfer the memory modules to a computer he controls and use it to extract their contents. Cooling the memory before powering it off slows the decay sufficiently to allow it to be transplanted with minimal data loss. As shown in Figure 5, widely available "canned air" dusting spray can be used to cool the chips to –50°C and below. At these temperatures data can be recovered with low error rates even after several minutes.

4. KEY RECONSTRUCTION

The attacker's task is more complicated when the memory is partially decayed, since there may be errors in the cryptographic keys he extracts, but we find that attacks can remain practical. We have developed algorithms for correcting errors in symmetric and private keys that can efficiently reconstruct keys when as few as 27% of the bits are known, depending on the type of key.

Our algorithms achieve significantly better performance than brute force by considering information other than the actual key. Most cryptographic software is optimized by storing data precomputed from the key, such as a key schedule for block ciphers or an extended form of the private key for RSA. This data contains much more structure than the key itself, and we can use this structure to perform efficient error correction.

These results imply a trade-off between efficiency and security. All of the disk encryption systems we studied precompute key schedules and keep them in memory for as long as the encrypted disk is mounted. While this practice saves some computation for each disk access, we find that it also facilitates attacks.

Our algorithms make use of the fact that most decay is *unidirectional*. In our experiments, almost all bits decayed to a predictable ground state with only a tiny fraction flipping in the opposite direction. In practice, the probability of decaying to the ground state approaches 1 as time goes on, while the probability of flipping in the opposite direction remains tiny—less than 0.1% in our tests. We further assume that the ground state decay probability is known to the attacker; it can be approximated by comparing the fractions of zeros and ones in the extracted key data and assuming that these were roughly equal before the data decayed.

4.1. Reconstructing DES keys

We begin with a relatively simple application of these ideas: an error-correction technique for DES keys. Before software can encrypt or decrypt data with DES, it must expand the secret key K into a set of *round keys* that are used internally by the cipher. The set of round keys is called the *key schedule*; since it takes time to compute, programs typically cache it in memory as long as K is in use. The DES key schedule consists of 16 round keys, each a permutation of a 48-bit subset of bits from the original 56-bit key. Every bit from the key is repeated in about 14 of the 16 round keys.

We begin with a partially decayed DES key schedule. For each bit of the key, we consider the n bits extracted from memory that were originally all identical copies of that key bit. Since we know roughly the probability that each bit decayed $0 \rightarrow 1$ or $1 \rightarrow 0$, we can calculate whether the extracted bits were more likely to have resulted from the decay of reptitions of 0 or repetitions of 1.

If 5% of the bits in the key schedule have decayed to the ground state, the probability that this technique will get any of the 56 bits of the key wrong is less than 10^{-8}. Even if 25% of

Figure 5: Advanced cold-boot attack. In our most powerful attack, the attacker reduces the temperature of the memory chips while the computer is still running, then physically moves them to another machine configured to read them without overwriting any data. Before powering off the computer, the attacker can spray the chips with "canned air," holding the container in an inverted position so that it discharges cold liquid refrigerant instead of gas (*left*). This cools the chips to around –50°C (*middle*). At this temperature, the data will persist for several minutes after power loss with minimal error, even if the memory modules are removed from the computer (*right*).

the bits in the key schedule are in error, the probability that we can correctly reconstruct the key without resorting to a brute force search is more than 98%.

4.2. Reconstructing AES keys

AES is a more modern cipher than DES, and it uses a key schedule with a more complex structure, but nevertheless we can efficiently reconstruct keys. For 128-bit keys, the AES key schedule consists of 11 round keys, each made up of four 32-bit words. The first round key is equal to the key itself. Each subsequent word of the key schedule is generated either by XORing two earlier words, or by performing an operation called the key schedule core (in which the bytes of a word are rotated and each byte is mapped to a new value) on an earlier word and XORing the result with another earlier word.

Instead of trying to correct an entire key at once, we can examine a smaller set of the bits at a time and then combine the results. This separability is enabled by the high amount of linearity in the key schedule. Consider a "slice" of the first two round keys consisting of byte i from words 1 to 3 of the first two round keys, and byte $i - 1$ from word 4 of the first round key (see Figure 6). This slice is 7 bytes long, but it is uniquely determined by the 4 bytes from the first round key.

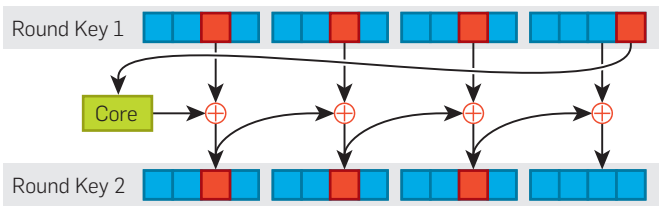
Our algorithm exploits this fact as follows. For each possible set of 4 key bytes, we generate the relevant 3 bytes of the next round key, and we order these possibilities by the likelihood that these 7 bytes might have decayed to the corresponding bytes extracted from memory. Now we may recombine four slices into a candidate key, in order of decreasing likelihood. For each candidate key, we calculate the key schedule. If the likelihood of this key schedule decaying to the bytes we extracted from memory is sufficiently high, we output the corresponding key.

When the decay is largely unidirectional, this algorithm will almost certainly output a unique guess for the key. This is because a single flipped bit in the key results in a cascade of bit flips through the key schedule, half of which are likely to flip in the "wrong" direction.

Our implementation of this algorithm is able to reconstruct keys with 7% of the bits decayed in a fraction of a second. It succeeds within 30 s for about half of keys with 15% of bits decayed.

We have extended this idea to 256-bit AES keys and to other ciphers. See the full paper for details.

Figure 6: Error correction for AES keys. In the AES-128 key schedule, 4 bytes from each round key completely determine 3 bytes of the next round key, as shown here. Our error correction algorithm "slices" the key into four groups of bytes with this property. It computes a list of likely candidate values for each slice, then checks each combination to see if it is a plausible key.



4.3. Reconstructing RSA private keys

An RSA public key consists of the modulus N and the public exponent e, while the private key consists of the private exponent d and several optional values: prime factors p and q of N, $d \bmod (p-1)$, $d \bmod (q-1)$, and $q^{-1} \bmod p$. Given N and e, any of the private values is sufficient to efficiently generate the others. In practice, RSA implementations store some or all of these values to speed computation.

In this case, the structure of the key information is the mathematical relationship between the fields of the public and private key. It is possible to iteratively enumerate potential RSA private keys and prune those that do not satisfy these relationships. Subsequent to our initial publication, Heninger and Shacham[11] showed that this leads to an algorithm that is able to recover in seconds an RSA key with all optional fields when only 27% of the bits are known.

5. IDENTIFYING KEYS IN MEMORY

After extracting the memory from a running system, an attacker needs some way to locate the cryptographic keys. This is like finding a needle in a haystack, since the keys might occupy only tens of bytes out of gigabytes of data. Simple approaches, such as attempting decryption using every block of memory as the key, are intractable if the memory contains even a small amount of decay.

We have developed fully automatic techniques for locating encryption keys in memory images, even in the presence of errors. We target the key schedule instead of the key itself, searching for blocks of memory that satisfy the properties of a valid key schedule.

Although previous approaches to key recovery do not require a key schedule to be present in memory, they have other practical drawbacks that limit their usefulness for our purposes. Shamir and van Someren[16] conjecture that keys have higher entropy than the other contents of memory and claim that they should be distinguishable by a simple visual test. However, even perfect copies of memory often contain large blocks of random-looking data (e.g., compressed files). Pettersson[15] suggests locating program data structures containing key material based on the range of likely values for each field. This approach requires the manual derivation of search heuristics for each cryptographic application, and it is not robust to memory errors.

We propose the following algorithm for locating scheduled AES keys in extracted memory:

1. Iterate through each byte of memory. Treat that address as the start of an AES key schedule.
2. Calculate the Hamming distance between each word in the potential key schedule and the value that would have been generated from the surrounding words in a real, undecayed key schedule.
3. If the sum of the Hamming distances is sufficiently low, the region is close to a correct key schedule; output the key.

We implemented this algorithm for 128- and 256-bit AES keys in an application called `keyfind`. The program receives extracted memory and outputs a list of likely keys. It assumes

that key schedules are contiguous regions of memory in the byte order used in the AES specification; this can be adjusted for particular cipher implementations. A threshold parameter controls how many bit errors will be tolerated.

As described in Section 6, we successfully used `keyfind` to recover keys from closed-source disk encryption programs without having to reverse engineer their key data structures. In other tests, we even found key schedules that were partially overwritten after the memory where they were stored was reallocated.

This approach can be applied to many other ciphers, including DES. To locate RSA keys, we can search for known key data or for characteristics of the standard data structure used for storing RSA private keys; we successfully located the SSL private keys in memory extracted from a computer running Apache 2.2.3 with `mod_ssl`. For details, see the full version of this paper.

6. ATTACKING ENCRYPTED DISKS

We have applied the tools developed in this paper to defeat several popular on-the-fly disk encryption systems, and we suspect that many similar products are also vulnerable. Our results suggest that disk encryption, while valuable, is not necessarily a sufficient defense against physical data theft.

6.1. BitLocker

BitLocker is a disk encryption feature included with some versions of Windows Vista and Windows 7. It operates as a filter driver that resides between the file system and the disk driver, encrypting and decrypting individual sectors on demand. As described in a paper by Niels Ferguson of Microsoft,[8] the BitLocker encryption algorithm encrypts data on the disk using a pair of AES keys, which, we discovered, reside in RAM in scheduled form for as long as the disk is mounted.

We created a fully automated demonstration attack tool called Bit*Un*locker. It consists of an external USB hard disk containing a Linux distribution, a custom SYSLINUX-based bootloader, and a custom driver that allows BitLocker volumes to be mounted under Linux. To use it against a running Windows system, one cuts power momentarily to reset the machine, then connects the USB disk and boots from the external drive. BitUnlocker automatically dumps the memory image to the external disk, runs `keyfind` to locate candidate keys, tries all combinations of the candidates, and, if the correct keys are found, mounts the BitLocker encrypted volume. Once the encrypted volume has been mounted, one can browse it using the Linux distribution just like any other volume.

We tested this attack on a modern laptop with 2GB of RAM. We rebooted it by removing the battery and cutting power for less than a second; although we did not use any cooling, BitUnlocker successfully recovered the keys with no errors and decrypted the disk. The entire automated process took around 25 min, and optimizations could greatly reduce this time.

6.2. FileVault

Apple's FileVault disk encryption software ships with recent versions of Mac OS X. A user-supplied password decrypts a header that contains both an AES key used to encrypt stored data and a second key used to compute IVs (initialization vectors).[18]

We used our EFI memory extraction program on an Intel-based Macintosh system running Mac OS X 10.4 with a FileVault volume mounted. Our `keyfind` program automatically identified the FileVault AES encryption key, which did not contain any bit errors in our tests.

As for the IV key, it is present in RAM while the disk is mounted, and if none of its bits decay, an attacker can identify it by attempting decryption using all appropriately sized substrings of memory. FileVault encrypts each disk block in CBC (cipher-block chaining) mode, so even if the attacker cannot recover the IV key, he can decrypt 4080 bytes of each 4096 byte disk block (all except the first cipher block) using only the AES key. The AES and IV keys together allow full decryption of the volume using programs like `vilefault`.[18]

6.3. TrueCrypt, dm-crypt, and Loop-AES

We tested three popular open-source disk encryption systems, TrueCrypt, dm-crypt, and Loop-AES, and found that they too are vulnerable to attacks like the ones we have described. In all three cases, once we had extracted a memory image with our tools, we were able to use `keyfind` to locate the encryption keys, which we then used to decrypt and mount the disks.

7. COUNTERMEASURES

Memory remanence attacks are difficult to prevent because cryptographic keys in active use must be stored *somewhere*. Potential countermeasures focus on discarding or obscuring encryption keys before an adversary might gain physical access, preventing memory extraction software from executing on the machine, physically protecting the DRAM chips, and making the contents of memory decay more readily.

7.1. Suspending a system safely

Simply locking the screen of a computer (i.e., keeping the system running but requiring entry of a password before the system will interact with the user) does not protect the contents of memory. Suspending a laptop's state to RAM (sleeping) is also ineffective, even if the machine enters a screen-locked state on awakening, since an adversary could simply awaken the laptop, power-cycle it, and then extract its memory state. Suspending to disk (hibernating) may also be ineffective unless an externally held secret key is required to decrypt the disk when the system is awakened.

With most disk encryption systems, users can protect themselves by powering off the machine completely when it is not in use then guarding the machine for a minute or so until the contents of memory have decayed sufficiently. Though effective, this countermeasure is inconvenient, since the user will have to wait through the lengthy boot process before accessing the machine again.

Suspending can be made safe by requiring a password or other external secret to reawaken the machine and encrypting the contents of memory under a key derived from the password. If encrypting all of the memory is too expensive, the system could encrypt only those pages or regions containing important keys. An attacker might still try to guess the password and check his guesses by attempting decryption (an offline password-guessing attack), so systems

should encourage the use of strong passwords and employ password strengthening techniques[2] to make checking guesses slower. Some existing systems, such as Loop-AES, can be configured to suspend safely in this sense, although this is usually not the default behavior.

7.2. Storing keys differently

Our attacks show that using precomputation to speed cryptographic operations can make keys more vulnerable, because redundancy in the precomputed values helps the attacker reconstruct keys in the presence of memory errors. To mitigate this risk, implementations could avoid storing precomputed values, instead recomputing them as needed and erasing the computed information after use. This improves resistance to memory remanence attacks but can carry a significant performance penalty. (These performance costs are negligible compared to the access time of a hard disk, but disk encryption is often implemented on top of disk caches that are fast enough to make them matter.)

Implementations could transform the key as it is stored in memory in order to make it more difficult to reconstruct in the case of errors. This problem has been considered from a theoretical perspective; Canetti et al.[3] define the notion of an *exposure-resilient function* (ERF) whose input remains secret even if all but some small fraction of the output is revealed. This carries a performance penalty because of the need to reconstruct the key before using it.

7.3. Physical defenses

It may be possible to physically defend memory chips from being removed from a machine, or to detect attempts to open a machine or remove the chips and respond by erasing memory. In the limit, these countermeasures approach the methods used in secure coprocessors[7] and could add considerable cost to a PC. However, a small amount of memory soldered to a motherboard would provide moderate defense for sensitive keys and could be added at relatively low cost.

7.4. Architectural changes

Some countermeasures involve changes to the computer's architecture that might make future machines more secure. DRAM systems could be designed to lose their state quickly, though this might be difficult, given the need to keep the probability of decay within a DRAM refresh interval vanishingly small. Key-store hardware could be added—perhaps inside the CPU—to store a few keys securely while erasing them on power-up, reset, and shutdown. Some proposed architectures would routinely encrypt the contents of memory for security purposes[6, 12]; these would prevent the attacks we describe as long as the keys are reliably destroyed on reset or power loss.

7.5. Encrypting in the disk controller

Another approach is to perform encryption in the disk controller rather than in software running on the main CPU and to store the key in the controller's memory instead of the PC's DRAM. In a basic form of this approach, the user supplies a secret to the disk at boot, and the disk controller uses this secret to derive a symmetric key that it uses to encrypt and decrypt the disk contents.

For this method to be secure, the disk controller must erase the key from its memory whenever the computer is rebooted. Otherwise, an attacker could reboot into a malicious kernel that simply reads the disk contents. For similar reasons, the key must also be erased if an attacker attempts to transplant the disk to another computer.

While we leave an in-depth study of encryption in the disk controller to future work, we did perform a cursory test of two hard disks with this capability, the Seagate Momentus 5400 FDE.2 and the Hitachi 7K200. We found that they do not appear to defend against the threat of transplantation. We attached both disks to a PC and confirmed that every time we powered on the machine, we had to enter a password via the BIOS in order to decrypt the disks. However, once we had entered the password, we could disconnect the disks' SATA cables from the motherboard (leaving the power cables connected), connect them to another PC, and read the disks' contents on the second PC without having to re-enter the password.

7.6. Trusted computing

Though useful against some attacks, most Trusted Computing hardware deployed in PCs today does not prevent the attacks described here. Such hardware generally does not perform bulk data encryption itself; instead, it monitors the boot process to decide (or help other machines decide) whether it is safe to store a key in RAM. If a software module wants to safeguard a key, it can arrange that the usable form of that key will not be stored in RAM unless the boot process has gone as expected. However, once the key is stored in RAM, it is subject to our attacks. Today's Trusted Computing devices can prevent a key from being loaded into memory for use, but they cannot prevent it from being captured once it is in memory.

In some cases, Trusted Computing makes the problem worse. BitLocker, in its default "basic mode," protects the disk keys solely with Trusted Computing hardware. When the machine boots, BitLocker automatically loads the keys into RAM from the Trusted Computing hardware without requiring the user to enter any secrets. Unlike other disk encryption systems we studied, this configuration is at risk even if the computer has been shut down for a long time—the attacks only needs to power on the machine to have the keys loaded back into memory, where they are vulnerable to our attacks.

8. PREVIOUS WORK

We owe the suggestion that DRAM contents can survive cold boot to Pettersson,[15] who seems to have obtained it from Chow et al.[5] Pettersson suggested that remanence across cold boot could be used to acquire forensic memory images and cryptographic keys. Chow et al. discovered the property during an unrelated experiment, and they remarked on its security implications. Neither experimented with those implications.

MacIver stated in a presentation[14] that Microsoft considered memory remanence in designing its BitLocker disk encryption system. He acknowledged that BitLocker is vulnerable to having keys extracted by cold-booting a machine when used in a "basic mode," but he asserted that BitLocker is not vulnerable in "advanced modes" (where a user must

provide key material to access the volume). MacIver apparently has not published on this subject.

Researchers have known since the 1970s that DRAM cell contents survive to some extent even at room temperature and that retention times can be increased by cooling.[13] In 2002, Skorobogatov[17] found significant retention times with *static* RAMs at room temperature. Our results for DRAMs show even longer retention in some cases.

Some past work focuses on "burn-in" effects that occur when data is stored in RAM for an extended period. Gutmann[9,10] attributes burn-in to physical changes in memory cells, and he suggests that keys be relocated periodically as a defense. Our findings concern a different phenomenon. The remanence effects we studied occur even when data is stored only momentarily, and they result not from physical changes but from the electrical capacitance of DRAM cells.

A number of methods exist for obtaining memory images from live systems. Unlike existing techniques, our attacks do not require access to specialized hardware or a privileged account on the target system, and they are resistant to operating system countermeasures.

9. CONCLUSION

Contrary to common belief, DRAMs hold their values for surprisingly long intervals without power or refresh. We show that this fact enables attackers to extract cryptographic keys and other sensitive information from memory despite the operating system's efforts to secure memory contents. The attacks we describe are practical—for example, we have used them to defeat several popular disk encryption systems. These results imply that disk encryption on laptops, while beneficial, does not guarantee protection.

In recent work Chan et al.[4] demonstrate a dangerous extension to our attacks. They show how to cold-reboot a running computer, surgically alter its memory, and then restore the machine to its previous running state. This allows the attacker to defeat a wide variety of security mechanisms—including disk encryption, screen locks, and antivirus software—by tampering with data in memory before reanimating the machine. This attack can potentially compromise data beyond the local disk; for example, it can be executed quickly enough to bypass a locked screen before any active VPN connections time out. Though it appears that this attack would be technically challenging to execute, it illustrates that memory's vulnerability to physical attacks presents serious threats that security researchers are only beginning to understand.

There seems to be no easy remedy for memory remanence attacks. Ultimately, it might become necessary to treat DRAM as untrusted and to avoid storing sensitive data there, but this will not be feasible until architectures are changed to give running software a safe place to keep secrets.

Acknowledgments

We thank Andrew Appel, Jesse Burns, Grey David, Laura Felten, Christian Fromme, Dan Good, Peter Gutmann, Benjamin Mako Hill, David Hulton, Brie Ilenda, Scott Karlin, David Molnar, Tim Newsham, Chris Palmer, Audrey Penven, David Robinson, Kragen Sitaker, N.J.A. Sloane, Gregory Sutter, Sam Taylor, Ralf-Philipp Weinmann, and Bill Zeller for their helpful contributions. This work was supported in part by a National Science Foundation Graduate Research Fellowship and by the Department of Homeland Security Scholarship and Fellowship Program; it does not necessarily reflect the views of NSF or DHS.

References

1. Arbaugh, W., Farber, D., Smith, J. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 1997), 65–71.
2. Boyen, X. Halting password puzzles: Hard-to-break encryption from human-memorable keys. In *Proceedings of the 16th USENIX Security Symposium* (August 2008).
3. Canetti, R., Dodis, Y., Halevi, S., Kushilevitz, E., Sahai, A. Exposure-resilient functions and all-or-nothing transforms. In *EUROCRYPT 2000*, volume 1807/2000 (2000), 453–469.
4. Chan, E.M., Carlyle, J.C., David, F.M., Farivar, R., Campbell, R.H. Bootjacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (October 2008), 555–564.
5. Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium* (August 2005), 331–346.
6. Dwoskin, J., Lee, R.B. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (October 2007), 389–400.
7. Dyer, J.G., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S.W., Weingart, S. Building the IBM 4758 secure coprocessor. *Computer 34* (Oct. 2001), 57–66.
8. Ferguson, N. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista, (August 2006).
9. Gutmann, P. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium* (July 1996), 77–90.
10. Gutmann, P. Data remanence in semiconductor devices. In *Proceedings of the 10th USENIX Security Symposium* (August 2001), 39–54.
11. Heninger, N., Shacham, H. Improved RSA private key reconstruction for cold boot attacks. Cryptology ePrint Archive, Report 2008/510, December 2008.
12. Lie, D., Thekkath, C.A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M. Architectural support for copy and tamper resistant software. In *Symposium on Architectural Support for Programming Languages and Operating Systems* (2000).
13. Link, W., May, H. Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen. *Archiv für Elektronik und Übertragungstechnik 33* (June 1979), 229–235.
14. MacIver, D. Penetration testing Windows Vista BitLocker drive encryption. Presentation, Hack In The Box (September 2006).
15. Pettersson, T. Cryptographic key recovery from Linux memory dumps. Presentation, Chaos Communication Camp (August 2007).
16. Shamir, A., van Someren, N. Playing "hide and seek" with stored keys. *LNCS 1648* (1999), 118–124.
17. Skorobogatov, S. Low-temperature data remanence in static RAM. University of Cambridge Computer Laboratory Technical Report 536, June 2002.
18. Weinmann, R.-P., Appelbaum, J. Unlocking FileVault. Presentation, 23rd Chaos Communication Congress, December 2006.

J. Alex Halderman
(jhalderm@eecs.umich.edu)
University of Michigan.

Seth D. Schoen
(schoen@eff.org)
Electronic Frontier Foundation.

Nadia Heninger
(nadiah@cs.princeton.edu)
Princeton University.

William Clarkson
(wclarkso@cs.princeton.edu)
Princeton University.

William Paul
(wpaul@windriver.com)
Wind River Systems.

Joseph A. Calandrino
(jcalandr@cs.princeton.edu)
Princeton University.

Ariel J. Feldman
(ajfeldma@cs.princeton.edu)
Princeton University.

Jacob Appelbaum
(jacob@appelbaum.net)
The Tor Project.

Edward W. Felten
(felten@cs.princeton.edu)
Princeton University.

DOI:10.1145/1506409.1506430

Technical Perspective
Highly Concurrent Data Structures

By Maurice Herlihy

THE ADVENT OF multicore architectures has produced a Renaissance in the study of highly concurrent data structures. Think of these shared data structures as the ball bearings of concurrent architectures: they are the potential "hot spots" where concurrent threads synchronize. Under-engineered data structures, like under-engineered ball bearings, can prevent individually well-engineered parts from performing well together. Simplifying somewhat, Amdahl's Law states that synchronization granularity matters: even short sequential sections can hamstring the scalability of otherwise well-designed concurrent systems.

The design and implementation of libraries of highly concurrent data structures will become increasingly important as applications adapt to multicore platforms. Well-designed concurrent data structures illustrate the power of *abstraction*: On the outside, they provide clients with simple sequential specifications that can be understood and exploited by nonspecialists. For example, a data structure might simply describe itself as a map from keys to values. An operation such as inserting a key-value binding in the map appears to happen instantaneously in the interval between when the operation is called and when it returns, a property known as *linearizability*. On the inside, however, they may be highly engineered by specialists to match the characteristics of the underlying platform.

Scherer, Lea, and Scott's "Scalable Synchronous Queues" is a welcome addition to a growing repertoire of scalable concurrent data structures. *Communications'* Research Highlights editorial board chose this paper for several reasons. First, it is a useful algorithm in its own right. Moreover, it is the very model of a modern concurrent data structures paper. The interface is simple, the internal structure, while clever, is easily understood, the correctness arguments are concise and clear. It provides a small number of useful choices, such as the ability to time out or to trade performance for fairness, and the experimental validation is well described and reproducible.

This synchronous queue is *lock-free*: the delay or failure of one thread cannot delay others from completing that operation. There are three principal nonblocking progress properties in the literature. An operation is *wait-free* if all threads calling that operation will eventually succeed. It is *lock-free* if some thread will succeed, and it is *obstruction-free* if some thread will succeed provided no conflicting thread runs at the same time. Note that a data structure may provide different guarantees for different operations: a map might provide lock-free insertion but wait-free lookups. In practice, most non-blocking algorithms are lock-free.

Writing lock-free algorithms, like writing device drivers and cosine routines, requires some care and expertise.

Lock-free operations are attractive for several reasons. They are robust against unexpected delays. In modern multicore architectures, threads are subject to long and unpredictable delays, ranging from cache misses (short), signals (long), page faults (very long), to being descheduled (very, very long). For example, if a thread is holding a lock when it is descheduled, then other, running threads that need that lock will also be blocked. With locks, systems with real-time constraints may be subject to *priority inversion*, where a high-priority thread is blocked waiting for a low-priority thread to release a lock. Care must be taken to avoid deadlocks, where threads wait forever for one another to release locks.

Amdahl's Law says that the shorter the critical sections, the better. One can think of lock-free synchronization as a limiting case of this trend, reducing critical sections to individual machine instructions. As a result, however, lock-free algorithms are often tricky to implement. The need to avoid overhead can lead to complicated designs, which may in turn make it difficult to reason (even informally) about correctness. Nevertheless, lock-free algorithms are not necessarily more difficult than other kinds of highly concurrent algorithms. Writing lock-free algorithms, like writing device drivers or cosine routines, requires some care and expertise.

Given such difficulty, can lock-free synchronization live up to its promise? In fact, lock-free synchronization has had a number of success stories. Widely used packages such as Java's java.util.concurrent, and C#'s System.Threading.Collections include a variety of finely tuned lock-free data structures. Applications that have benefited from lock-free data structures fall into categories as diverse as work-stealing schedulers, memory allocation programs, operating systems, music, and games.

For the foreseeable future, concurrent data structures will lie at the heart of multicore applications, and the larger our library of scalable concurrent data structures, the better we can exploit the promise of multicore architectures. C

Maurice Herlihy is a professor of computer science at Brown University, Providence, R.I. He is the recipient of the 2004 Gödel Prize and the 2003 Dijkstra Prize and is a member of the editorial board for *Communications*' Research Highlights section.

DOI:10.1145/1506409.1506431

Scalable Synchronous Queues

By William N. Scherer III, Doug Lea, and Michael L. Scott

Abstract

In a thread-safe *concurrent queue*, consumers typically wait for producers to make data available. In a *synchronous queue*, producers similarly wait for consumers to take the data. We present two new nonblocking, contention-free synchronous queues that achieve high performance through a form of *dualism*: The underlying data structure may hold both data and, symmetrically, *requests*.

We present performance results on 16-processor SPARC and 4-processor Opteron machines. We compare our algorithms to commonly used alternatives from the literature and from the Java SE 5.0 class *java.util.concurrent.SynchronousQueue* both directly in synthetic microbenchmarks and indirectly as the core of Java's *ThreadPoolExecutor* mechanism. Our new algorithms consistently outperform the Java SE 5.0 *SynchronousQueue* by factors of three in unfair mode and 14 in fair mode; this translates to factors of two and ten for the *ThreadPoolExecutor*. Our synchronous queues have been adopted for inclusion in Java 6.

1. INTRODUCTION

Mechanisms to transfer data between threads are among the most fundamental building blocks of concurrent systems. Shared memory transfers are typically effected via a concurrent data structure that may be known variously as a *buffer*, a *channel*, or a *concurrent queue*. This structure serves to "pair up" producers and consumers. It can also serve to smooth out fluctuations in their relative rates of progress by buffering unconsumed data. This buffering, in systems that provide it, is naturally asymmetric: A consumer that tries to `take` data from an empty concurrent queue will wait for a producer to perform a matching `put` operation; however, a producer need not wait to perform a `put` unless space has run out. That is, producers can "run ahead" of consumers, but consumers cannot "run ahead" of producers.

A *synchronous queue* provides the "pairing up" function without the buffering; it is entirely symmetric: Producers and consumers wait for one another, "shake hands," and leave in pairs. For decades, synchronous queues have played a prominent role in both the theory and practice of concurrent programming. They constitute the central synchronization primitive of Hoare's CSP[8] and of languages derived from it, and are closely related to the *rendezvous* of Ada. They are also widely used in message-passing software and in stream-style "hand-off" algorithms.[2, Chap. 8] (In this paper we focus on synchronous queues within a multithreaded program, not across address spaces or distributed nodes.)

Unfortunately, design-level tractability of synchronous queues has often come at the price of poor performance. "Textbook" algorithms for `put` and `take` may repeatedly suffer from *contention* (slowdown due to conflicts with other threads for access to a cache line) and/or *blocking* (loops or scheduling operations that wait for activity in another thread). Listing 1, for example, shows one of the most commonly used implementations, due to Hanson.[3] It employs three separate *semaphores*, each of which is a potential source of contention and (in `acquire` operations) blocking.[a]

The synchronization burden of algorithms like Hanson's is especially significant on modern multicore and multiprocessor machines, where the OS scheduler may take thousands of cycles to block or unblock threads. Even an uncontended semaphore operation usually requires special read-modify-write or memory barrier (fence) instructions, each of which can take tens of cycles.[b]

Listing 1: Hanson's synchronous queue. Semaphore `sync` indicates whether `item` is valid (initially, no); `send` holds 1 minus the number of pending `puts`; `recv` holds 0 minus the number of pending `takes`.

```
00 public class HansonSQ<E> {
01   E item = null;
02   Semaphore sync = new Semaphore(0);
03   Semaphore send = new Semaphore(1);
04   Semaphore recv = new Semaphore(0);
05
06   Public E take() {
07     recv.acquire();
08     E x = item;
09     sync.release();
10     send.release();
11     return x;
12   }
13
14   public void put(E x) {
15     send.acquire();
16     item = x;
17     recv.release();
18     sync.acquire();
19   }
20 }
```

[a] Semaphores are the original mechanism for scheduler-based synchronization (they date from the mid-1960s). Each semaphore contains a counter and a list of waiting threads. An `acquire` operation decrements the counter and then waits for it to be nonnegative. A `release` operation increments the counter and unblocks a waiting thread if the result is nonpositive. In effect, a semaphore functions as a non-synchronous concurrent queue in which the transferred data is null.

[b] Read-modify-write instructions (e.g., `compare_and_swap` [CAS]) facilitate constructing concurrent algorithms via atomic memory updates. Fences enforce ordering constraints on memory operations.

A previous version of this paper was published in *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

It is also difficult to extend Listing 1 and other "classic" synchronous queue algorithms to provide additional functionality. Many applications require `poll` and `offer` operations, which `take` an item only if a producer is already present, or `put` an item only if a consumer is already waiting (otherwise, these operations return an error). Similarly, many applications require the ability to time out if producers or consumers do not appear within a certain *patience* interval or if the waiting thread is asynchronously interrupted. In the *java.util.concurrent* library, one of the *ThreadPoolExecutor* implementations uses all of these capabilities: Producers deliver tasks to waiting worker threads if immediately available, but otherwise create new worker threads. Conversely, worker threads terminate themselves if no work appears within a given keep-alive period (or if the pool is shut down via an interrupt).

Additionally, applications using synchronous queues vary in their need for *fairness*: Given multiple waiting producers, it may or may not be important to an application whether the one waiting the longest (or shortest) will be the next to pair up with the next arriving consumer (and vice versa). Since these choices amount to application-level policy decisions, algorithms should minimize imposed constraints. For example, while fairness is often considered a virtue, a thread pool normally runs faster if the most-recently-used waiting worker thread usually receives incoming work, due to the footprint retained in the cache and the translation lookaside buffer.

In this paper we present synchronous queue algorithms that combine a rich programming interface with very low intrinsic overhead. Our algorithms avoid all blocking other than that intrinsic to the notion of synchronous handoff: A producer thread must wait until a consumer appears (and vice versa); there is no other way for one thread's delay to impede another's progress. We describe two algorithmic variants: a *fair* algorithm that ensures strict FIFO ordering and an *unfair* algorithm that makes no guarantees about ordering (but is actually based on a LIFO stack). Section 2 of this paper presents the background for our approach. Section 3 describes the algorithms and Section 4 presents empirical performance data. We conclude and discuss potential extensions to this work in Section 5.

2. BACKGROUND

2.1. Nonblocking synchronization

Concurrent data structures are commonly protected with *locks*, which enforce *mutual exclusion* on *critical sections* executed by different threads. A naive synchronous queue might be protected by a single lock, forcing all `put` and `take` operations to execute serially. (A thread that blocked waiting for a peer would of course release the lock, allowing the peer to execute the matching operation.) With a bit of care and a second lock, we might allow one producer and one consumer to execute concurrently in many cases.

Unfortunately, locks suffer from several serious problems. Among other things, they introduce blocking beyond that required by data structure semantics: If thread A holds a lock that thread B needs, then B must wait, even if A has been preempted and will not run again for quite a while. A multiprogrammed system with thread priorities or asynchronous events may suffer spurious deadlocks due to *priority inversion*: B needs the lock A holds, but A cannot run, because B is a handler or has higher priority.

Nonblocking concurrent objects address these problems by avoiding mutual exclusion. Loosely speaking, their methods ensure that the object's invariants hold after every single instruction, and that its state can safely be seen—and manipulated—by other concurrent threads. Unsurprisingly, devising such methods can be a tricky business, and indeed the number of data structures for which correct nonblocking implementations are known is fairly small.

Linearizability[7] is the standard technique for demonstrating that a nonblocking implementation of an object is *correct* (i.e., that it continuously maintains object invariants). Informally, linearizability "provides the illusion that each operation... takes effect instantaneously at some point between its invocation and its response."[7, abstract] Orthogonally, nonblocking implementations may provide guarantees of various strength regarding the *progress* of method calls. In a *wait-free* implementation, every contending thread is guaranteed to complete its method call within a bounded number of its own execution steps.[5] Wait-free algorithms tend to have unacceptably high overheads in practice, due to the need to finish operations on other threads' behalf. In a *lock-free* implementation, *some* contending thread is guaranteed to complete its method call within a bounded number of any thread's steps.[5] The algorithms we present in this paper are all lock-free. Some algorithms provide a weaker guarantee known as *obstruction freedom*; it ensures that a thread can complete its method call within a bounded number of steps in the absence of contention, i.e., if no other threads execute competing methods concurrently.[6]

2.2. Dual data structures

In traditional nonblocking implementations of concurrent objects, every method is total: It has no preconditions that must be satisfied before it can complete. Operations that might normally block before completing, such as dequeuing from an empty queue, are generally *totalized* to simply return a failure code when their preconditions are not met. By calling the totalized method in a loop until it succeeds, one can simulate the partial operation. This simulation, however, does not necessarily respect our intuition for object semantics. For example, consider the following sequence of events for threads A, B, C, and D:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
B's call returns the 1
A's call returns the 2

If thread A's call to dequeue is known to have started before thread B's call, then intuitively, we would think that A should get the first result out of the queue. Yet, with the call-in-a-loop idiom, ordering is simply a function of which

thread happens to retry its dequeue operation first once data becomes available. Further, each invocation of the totalized method introduces performance-degrading contention for memory–interconnect bandwidth.

As an alternative, suppose we could register a request for a hand-off partner. Inserting this *reservation* could be done in a nonblocking manner, and checking to see whether a partner has arrived to *fulfill* our reservation could consist of reading a Boolean flag in the request data structure. A *dual data structure*[16, 19] takes precisely this approach: Objects may contain both data and reservations. We divide partial methods into separate, first-class *request* and *follow-up* operations, each of which has its own invocation and response. A total queue, for example, would provide `dequeue_request` and `dequeue_followup` methods (Listing 2). By analogy with Lamport's bakery algorithm,[10] the request operation returns a unique *ticket* that represents the reservation and is then passed as an argument to the follow-up method. The follow-up, for its part, returns either the desired result (if one is *matched* to the ticket) or, if the method's precondition has not yet been satisfied, an error indication.

The key difference between a dual data structure and a "totalized" partial method is that linearization of the `p_request` call allows the dual data structure to determine the fulfillment order for pending requests. In addition, unsuccessful follow-ups, unlike unsuccessful calls to totalized methods, are readily designed to avoid bus or memory contention. For programmer convenience, we provide `demand` methods, which wait until they can return successfully. Our implementations use both busy-wait spinning and scheduler-based suspension to effect waiting in threads whose preconditions are not met.

When reasoning about progress, we must deal with the fact that a partial method may wait for an arbitrary amount of time (perform an arbitrary number of unsuccessful follow-ups) before its precondition is satisfied. Clearly it is desirable that requests and follow-ups be nonblocking. In practice, good system performance will also typically require that unsuccessful follow-ups not interfere with other threads' progress. We define a data structure as *contention-free* if none of its follow-up operations, in any execution, performs more than a constant number of remote memory accesses across all unsuccessful invocations with the same request ticket. On a machine with an invalidation-based cache coherence protocol, a read of location *o* by thread *t* is said to be *remote* if *o* has been written by some thread other than *t* since *t* last accessed it; a write by *t* is remote if *o* has been accessed by some thread other than *t* since *t* last wrote it. On a machine that cannot cache remote locations, an access is remote if it refers to memory allocated on another node. Compared to the *local-spin property*,[13] contention freedom allows operations to block in ways other than busy-wait spinning; in particular, it allows other actions to be performed while waiting for a request to be satisfied.

Listing 2: Combined operations: `dequeue` pseudocode (`enqueue` is symmetric).

```
datum dequeue(SynchronousQueue Q) {
  reservation r = Q.dequeue_reserve();
  do {
    datum d = Q.dequeue_followup(r);
    if (failed != d) return d;
    /* else delay -- spinning and/or scheduler-based */
  while (!timed_out());
  if (Q.dequeue_abort(r)) return failed;
  return Q.dequeue_followup(r);
}
```

3. ALGORITHM DESCRIPTIONS

In this section we discuss various implementations of synchronous queues. We start with classic algorithms used extensively in production software, then we review newer implementations that improve upon them. Finally, we describe our new algorithms.

3.1. Classic synchronous queues

Perhaps the simplest implementation of synchronous queues is the naive monitor-based algorithm that appears in Listing 3. In this implementation, a single monitor serializes access to a single item and to a `putting` flag that indicates whether a producer has currently supplied data. Producers wait for the flag to be clear (lines 15–16), set the flag (17), insert an item (18), and then wait until a consumer takes the data (20–21). Consumers await the presence of an item (05–06), take it (07), and mark it as taken (08) before returning. At each point where their actions might potentially unblock another thread, producer and consumer threads awaken all possible candidates (09, 20, 24). Unfortunately, this approach results in a number of wake-ups quadratic in the number of waiting producer and consumer threads; coupled with the high cost of blocking or

Listing 3: Naive synchronous queue.

```
00 public class NaiveSQ<E> {
01   boolean putting = false;
02   E item = null;
03
04   public synchronized E take() {
05     while (item == null)
06       wait();
07     E e = item;
08     item = null;
09     notifyAll();
10     return e;
11   }
12
13   public synchronized void put (E e) {
14     if (e == null) return;
15     while (putting)
16       wait();
17     putting = true;
18     item = e;
19     notifyAll();
20     while (item != null)
21       wait();
22     putting = false;
23     notifyAll();
24   }
25 }
```

unblocking a thread, this results in poor performance.

Hanson's synchronous queue (Listing 1) improves upon the naive approach by using semaphores to target wakeups to only the single producer or consumer thread that an operation has unblocked. However, as noted in Section 1, it still incurs the overhead of three separate synchronization events per transfer for each of the producer and consumer; further, it normally blocks at least once per operation. It is possible to streamline some of these synchronization points in common execution scenarios by using a fast-path acquire sequence;[11] this was done in early releases of the *dl.util.concurrent* package which evolved into *java.util.concurrent*.

3.2. The Java SE 5.0 synchronous queue

The Java SE 5.0 synchronous queue (Listing 4) uses a pair of queues (in fair mode; stacks for unfair mode) to separately hold waiting producers and consumers. This approach echoes the scheduler data structures of Anderson et al;.[1] it improves considerably on semaphore-based approaches. When a producer or consumer finds its counterpart already waiting, the new arrival needs to perform only one synchronization operation: acquiring a lock that protects both queues (line 18 or 33). Even if no counterpart is waiting, the only additional synchronization required is to await one (25 or 40). A transfer thus requires only three synchronization operations, compared to the six incurred by Hanson's algorithm. In particular, using a queue instead of a semaphore allows producers to publish data items as they arrive (line 36) instead of having to first awaken after blocking on a semaphore; consumers need not wait.

3.3. Combining dual data structures with synchronous queues

A key limitation of the Java SE 5.0 *SynchronousQueue* class is its reliance on a single lock to protect both queues. Coarse-grained synchronization of this form is well known for introducing serialization bottlenecks; by creating nonblocking implementations, we eliminate a major impediment to scalability.

Our new algorithms add support for time-out and for bidirectional synchronous waiting to our previous nonblocking dual queue and dual stack algorithms[19] (those in turn were derived from the classic Treiber stack[21] and the M&S queue[14]). The nonsynchronous dual data structures already block when a consumer arrives before a producer; our challenge is to arrange for producers to block until a consumer arrives as well. In the queue, waiting is accomplished by spinning until a pointer changes from null to non-null, or vice versa; in the stack, it is accomplished by pushing a "fulfilling" node and arranging for adjacent matching nodes to "annihilate" one another.

We describe basic versions of the synchronous dual queue and stack in the sections "The synchronous dual queue" and "The synchronous dual stack," respectively. The section "Time-out" then sketches the manner in which we add time-out support. The section "Pragmatics" discusses additional pragmatic issues. Throughout the discussion, we present fragments of code to illustrate particular features; full source is available online at http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/SynchronousQueue.java.

Listing 4: The Java SE 5.0 *SynchronousQueue* class, fair (queue-based) version. The unfair version uses stacks instead of queues, but is otherwise identical. (For clarity, we have omitted details of the way in which AbstractQueuedSynchronizers are used, and code to generalize waitingProducers and waitingConsumers to either stacks or queues.)

```
00 public class Java5SQ<E> {
01   ReentrantLock qlock = new ReentrantLock();
02   Queue waitingProducers = new Queue();
03   Queue waitingConsumers = new Queue();
04
05   static class Node
06     extends AbstractQueuedSynchronizer {
07     E item;
08     Node next;
09
10     Node(Object x) { item = x; }
11     void waitForTake() { /* (uses AQS) */ }
12     E waitForPut() { /* (uses AQS) */ }
13   }
14
15   public E take() {
16     Node node;
17     boolean mustWait;
18     qlock.lock();
19     node = waitingProducers.pop();
20     if(mustWait = (node == null))
21       node = waitingConsumers.push(null);
22     qlock.unlock();
23
24     if (mustWait)
25       return node.waitForPut();
26     else
27       return node.item;
28   }
29
30   public void put(E e) {
31     Node node;
32     boolean mustWait;
33     qlock.lock();
34     node = waitingConsumers.pop();
35     if (mustWait = (node == null))
36       node = waitingProducers.push(e);
37     qlock.unlock();
38
39     if (mustWait)
40       node.waitForTake();
41     else
42       node.item = e;
43   }
44 }
```

The Synchronous Dual Queue: We represent the synchronous dual queue as a singly linked list with head and tail pointers. The list may contain *data nodes* or *request nodes* (reservations), but never both at once. Listing 5 shows the `enqueue` method. (Except for the direction of data transfer, `dequeue` is symmetric.) To enqueue, we first read the head and tail pointers (lines 06–07). From here, there are two main cases. The first occurs when the queue is empty (`h == t`) or contains data (line 08). We read the next pointer for the tail-most node in the queue (09). If all values read are mutually consistent (10) and the queue's tail pointer is current (11), we attempt to insert our offering at the tail of the queue (13–14). If successful, we wait until a consumer signals that it has

Listing 5: Synchronous dual queue: Spin-based enqueue; `dequeue` is symmetric except for the direction of data transfer. The various `cas` *field* (`old,new`) operations attempt to change `field` from `old` to `new`, and return a success/failure indication. On modern processors they can be implemented with a single atomic `compare_and_swap` instruction, or its equivalent.

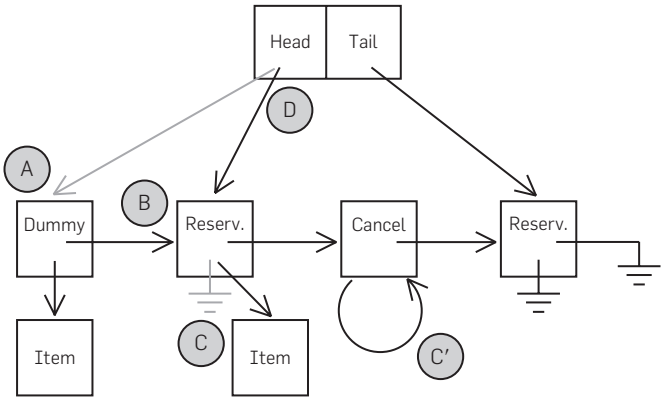
```
00 class Node { E data; Node next;...}
01
02 void enqueue(E e) {
03     Node offer = new Node(e, Data);
04
05     while (true) {
06         Node t = tail;
07         Node h = head;
08         if (h == t || !t.isRequest()) {
09             Node n = t.next;
10             if (t == tail) {
11                 if (null != n) {
12                   casTail(t, n);
13                 } else if(t.casNext(n, offer)) {
14                   casTail(t, offer);
15                   while (offer.data == e)
16                       /* spin */;
17                   h = head;
18                   if (offer == h.next)
19                   casHead(h, offer);
20                   return;
21                 }
22             }
23         } else {
24             Node n = h.next;
25             if (t != tail || h != head || n == null)
26                 continue; // inconsistent snapshot
27             boolean success = n.casData(null, e);
28             casHead(h, n);
29             if (success)
30                 return;
31         }
32     }
33 }
```

claimed our data (15–16), which it does by updating our node's data pointer to null. Then we help remove our node from the head of the queue and return (18–20). The request linearizes in this code path at line 13 when we successfully insert our offering into the queue; a successful follow-up linearizes when we notice at line 15 that our data has been taken.

The other case occurs when the queue consists of reservations, and is depicted in Figure 1. After originally reading the head node (step **A**), we read its successor (line 24/step **B**) and verify consistency (25). Then, we attempt to supply our data to the headmost reservation (27/**C**). If this succeeds, we dequeue the former dummy node (28/**D**) and return (30). If it fails, we need to go to the next reservation, so we dequeue the old dummy node anyway (28) and retry the entire operation (32, 05). The request linearizes in this code path when we successfully supply data to a waiting consumer at line 27; the follow-up linearization point occurs immediately thereafter.

The Synchronous Dual Stack: We represent the synchronous dual stack as a singly linked list with head pointer. Like the dual queue, the stack may contain either data or reservations, except that in this case there may, temporarily, be a single node of the opposite type at the head.

Figure 1: Synchronous dual queue: Enqueuing when reservations are present.



Listing 6: Synchronous dual stack: Spin-based annihilating push; `pop` is symmetric except for the direction of data transfer. (For clarity, code for time-out is omitted.)

```
00 class Node { E data; Node next, match; ... }
01
02 void push (E e) {
03     Node f, d = new Node(e, Data);
04
05     while (true) {
06         Node h = head;
07         if (null == h || h.isData()) {
08             d.next = h;
09             if (!casHead(h, d))
10                 continue;
11             while (d.match == null)
12                 /* spin */;
13             h = head;
14             if (null != h && d == h.next)
15                 casHead(h, d.next);
16             return;
17         } else if (h.isRequest()) {
18             f = new Node(e, Data | Fulfilling, h);
19             if (!casHead(h, f))
20                 continue;
21             h = f.next;
22             Node n = h.next;
23             h.casMatch(null, f);
24             casHead(f, n);
25             return;
26         } else { // h is fulfilling
27             Node n = h.next;
28             Node nn = n.next;
29             n.casMatch(null, h);
30             casHead(h, nn);
31         }
32     }
33 }
```

Code for the `push` operation appears in Listing 6. (Except for the direction of data transfer, `pop` is symmetric.) We begin by reading the node at the top of the stack (line 06).

The three main conditional branches (beginning at lines 07, 17, and 26) correspond to the type of node we find.

The first case occurs when the stack is empty or contains only data (line 07). We attempt to insert a new datum (09), and wait for a consumer to claim that datum (11–12) before returning. The reservation linearizes in this code path when we push our datum at line 09; a successful follow-up linearizes when we notice that our data has been taken at line 11.

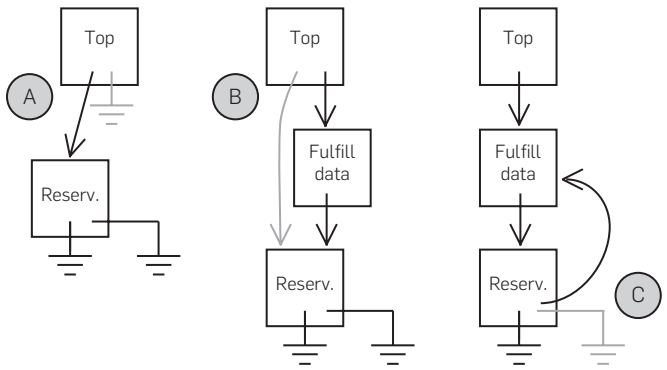
The second case occurs when the stack contains (only) reservations (17). We attempt to place a fulfilling datum on the top of the stack (19); if we succeed, any other thread that wishes to perform an operation must now help us fulfill the request before proceeding to its own work. We then read our way down the stack to find the successor node to the reservation we are fulfilling (21–22) and mark the reservation fulfilled (23). Note that our CAS could fail if another thread helps us and performs it first. Finally, we pop both the reservation and our fulfilling node from the stack (24) and return. The reservation linearizes in this code path at line 19, when we push our fulfilling datum above a reservation; the follow-up linearization point occurs immediately thereafter.

The remaining case occurs when we find another thread's fulfilling datum or reservation (26) at the top of the stack. We must complete the pairing and annihilation of the top two stack nodes before we can continue our own work. We first read our way down the stack to find the data or reservation for which the fulfilling node is present (27–28) and then we mark the underlying node as fulfilled (29) and pop the paired nodes from the stack (30).

Referring to Figure 2, when a consumer wishes to retrieve data from an empty stack, it first must insert a reservation (step **A**). It then waits until its data pointer (branching to the right) is non-null. Meanwhile, if a producer appears, it satisfies the consumer in a two-step process. First (step **B**), it pushes a fulfilling data node at the top of the stack. Then, it swings the reservation's data pointer to its fulfilling node (step **C**). Finally, it updates the top-of-stack pointer to match the reservation node's next pointer (step **D**, not shown). After the producer has completed step **B**, other threads can help update the reservation's data pointer (step **C**); and the consumer thread can additionally help remove itself from the stack (step **D**).

Time-Out: Although the algorithms presented in the sections "The Synchronous Dual Queue" and "The Synchronous Dual Stack" are complete implementations of synchronous queues, real systems require the ability to specify limited patience so that a producer (or consumer) can time out if no consumer (producer) arrives soon enough to pair up. As noted earlier, Hanson's synchronous queue offers no simple way to do this. Space limitations preclude discussion of the relatively straightforward manner in which we add time-out support to our synchronous queue; interested readers may find this information in our original publication.[17]

Figure 2: Synchronous dual stack: Satisfying a reservation.



Pragmatics: Our synchronous queue implementations reflect a few additional pragmatic considerations to maintain good performance. First, because Java does not allow us to set flag bits in pointers (to distinguish among the types of pointed-to nodes), we add an extra word to nodes, in which we mark mode bits. We chose this technique over two primary alternatives. The class *java.util.concurrent.AtomicMarkableReference* allows direct association of tag bits with a pointer, but exhibits very poor performance. Using runtime type identification (RTTI) to distinguish between multiple subclasses of the Node classes would similarly allow us to embed tag bits in the object type information. While this approach performs well in isolation, it increases long-term pressure on the JVM's memory allocation and garbage collection routines by requiring construction of a new node after each contention failure.

Time-out support requires careful management of memory ownership to ensure that canceled nodes are reclaimed properly. Automatic garbage collection eases the burden in Java. We must, however, take care to "forget" references to data, nodes, and threads that might be retained for a long time by blocked threads (preventing the garbage collector from reclaiming them).

The simplest approach to time-out involves marking nodes as "canceled," and abandoning them for another thread to eventually unlink and reclaim. If, however, items are offered at a very high rate, but with a very low time-out patience, this "abandonment" cleaning strategy can result in a long-term build-up of canceled nodes, exhausting memory supplies and degrading performance. It is important to effect a more sophisticated cleaning strategy. Space limitations preclude further discussion here, but interested readers may find more details in the conference version of this paper.[17]

For sake of clarity, the synchronous queues of Figures 5 and 6 blocked with busy-wait spinning to await a counterpart consumer. In practice, however, busy-wait is useless overhead on a uniprocessor and can be of limited value on even a small-scale multiprocessor. Alternatives include descheduling a thread until it is signaled, or yielding the processor within a spin loop.[9] In practice, we mainly choose the spin-then-yield approach, using the `park` and `unpark` methods contained in *java.util.concurrent.locks.LockSupport*[12] to remove threads from and restore threads to the ready list. On multiprocessors (only), nodes next in line for fulfillment spin briefly (about one-quarter the time of a typical context switch) before parking. On very busy synchronous queues, spinning can dramatically improve throughput because it handles the case of a near-simultaneous "flyby" between a producer and consumer without stalling either. On less busy

queues, the amount of spinning is small enough not to be noticeable.

4. EXPERIMENTAL RESULTS

We present results for several microbenchmarks and one "real-world" scenario. The microbenchmarks employ threads that produce and consume as fast as they can; this represents the limiting case of producer-consumer applications as the cost to process elements approaches zero. We consider producer-consumer ratios of 1 : *N*, *N* : 1, and *N* : *N*.

Our "real-world" scenario instantiates synchronous queues as the core of the Java SE 5.0 class *java.util.concurrent.ThreadPoolExecutor*, which in turn forms the backbone of many Java-based server applications. Our benchmark produces *tasks* to be run by a pool of worker threads managed by the *ThreadPoolExecutor*.

We obtained results on a SunFire V40z with four 2.4GHz AMD Opteron processors and on a SunFire 6800 with 16 1.3GHz Ultra-SPARC III processors. On both machines, we used Sun's Java SE 5.0 HotSpot VM and we varied the level of concurrency from 2 to 64. We tested each benchmark with both the fair and unfair (stack-based) versions of the Java SE 5.0 *java.util.concurrent.SynchronousQueue*, Hanson's synchronous queue, and our new nonblocking algorithms.

Figure 3 displays the rate at which data is transferred from multiple producers to multiple consumers; Figure 4 displays the rate at which data is transferred from a single producer to multiple consumers; Figure 5 displays the rate at which a single consumer receives data from multiple producers. Figure 6 presents execution time per task for our *ThreadPoolExecutor* benchmark.

As can be seen from Figure 3, Hanson's synchronous queue and the Java SE 5.0 fair-mode synchronous queue both perform relatively poorly, taking 4 (Opteron) to 8 (SPARC) times as long to effect a transfer relative to the faster algorithms. The unfair (stack-based) Java SE 5.0 synchronous queue in turn incurs twice the overhead of either the fair or unfair version of our new algorithm, both versions of which are comparable in performance. The main reason that the Java SE 5.0 fair-mode queue is so much slower than unfair is that the fair-mode version uses a fair-mode entry lock to ensure FIFO wait ordering. This causes pileups that block the threads that will fulfill waiting threads. This difference supports our claim that blocking and contention surrounding the synchronization state of synchronous queues are major impediments to scalability.

When a single producer struggles to satisfy multiple consumers (Figure 4), or a single consumer struggles to receive data from multiple producers (Figure 5), the disadvantages

Figure 3: Synchronous handoff: *N* producers, *N* consumers.

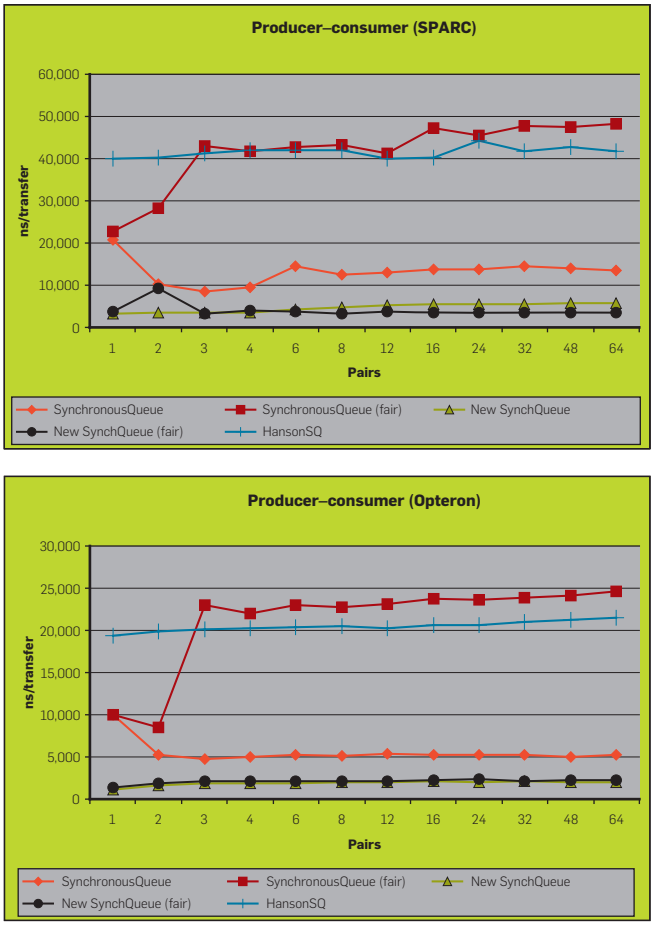


Figure 4: Synchronous handoff: 1 producer, *N* consumers.

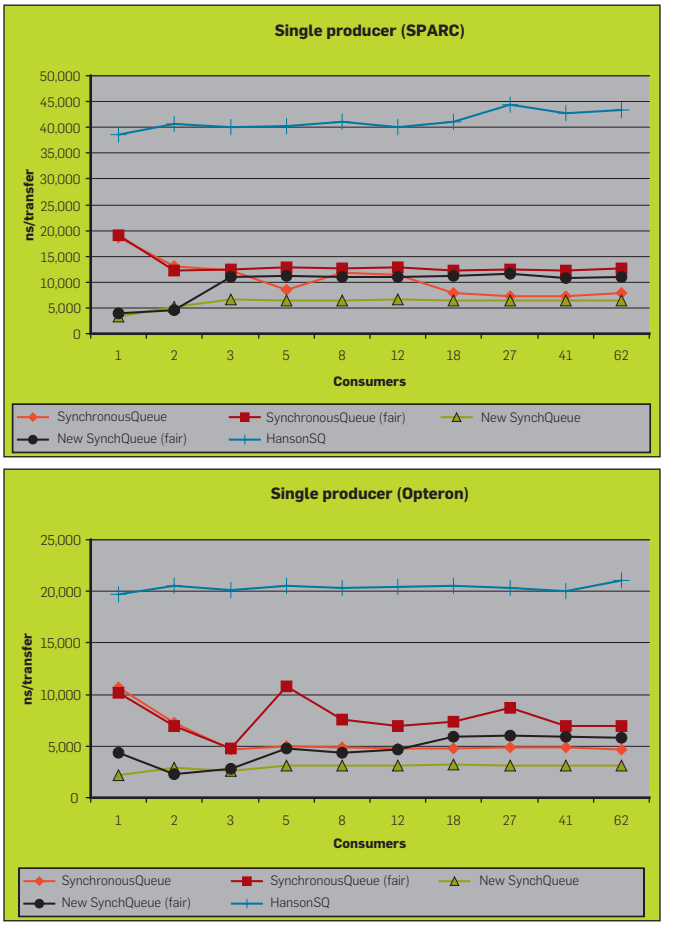


Figure 5: Synchronous handoff: *N* producers, 1 consumer.

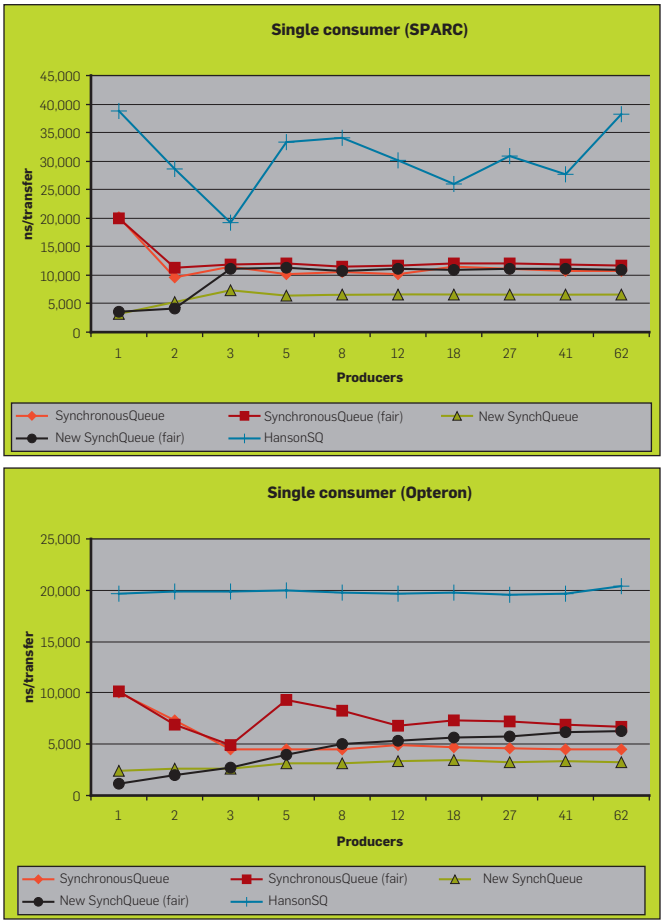
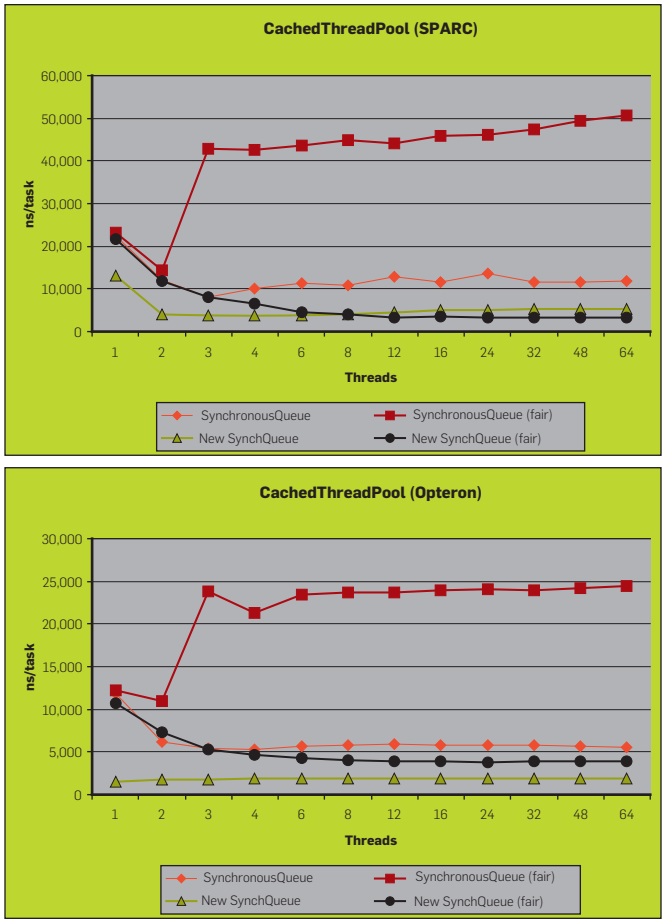


Figure 6: ThreadPoolExecutor benchmark.



of Hanson's synchronous queue are accentuated. Because the singleton necessarily blocks for every operation, the time it takes to produce or consume data increases noticeably. Our new synchronous queue consistently outperforms the Java SE 5.0 implementation (fair vs. fair and unfair vs. unfair) at all levels of concurrency.

Finally, in Figure 6, we see that the performance differentials from *java.util.concurrent*'s *SynchronousQueue* translate directly into overhead in the *ThreadPoolExecutor*: Our new fair version outperforms the Java SE 5.0 implementation by factors of 14 (SPARC) and 6 (Opteron); our unfair version outperforms Java SE 5.0 by a factor of three on both platforms. Interestingly, the relative performance of fair and unfair versions of our new algorithm differs between the two platforms. Generally, unfair mode tends to improve locality by keeping some threads "hot" and others buried at the bottom of the stack. Conversely, however, it tends to increase the number of times threads are scheduled and descheduled. On the SPARC, context switches have a higher relative overhead compared to other factors; this is why our fair synchronous queue eventually catches and surpasses the unfair version's performance. In contrast, the cost of context switches is relatively smaller on the Opteron, so the trade-off tips in favor of increased locality and the unfair version performs best.

Across all benchmarks, our fair synchronous queue universally outperforms all other fair synchronous queues and our unfair synchronous queue outperforms all other unfair synchronous queues, regardless of preemption or level of concurrency.

5. CONCLUSION

In this paper, we have presented two new lock-free and contention-free synchronous queues that outperform all previously known algorithms by a wide margin. In striking contrast to previous implementations, there is little performance cost for fairness.

In a head-to-head comparison, our algorithms consistently outperform the Java SE 5.0 *SynchronousQueue* by a factor of three in unfair mode and up to a factor of 14 in fair mode. We have further shown that this performance differential translates directly to factors of two and ten when substituting our new synchronous queue in for the core of the Java SE 5.0 *ThreadPoolExecutor*, which is itself at the heart of many Java-based server implementations. Our new synchronous queues have been adopted for inclusion in Java 6.

More recently, we have extended the approach described in this paper to *TransferQueues*. TransferQueues permit producers to enqueue data either synchronously or

asynchronously. TransferQueues are useful for example in supporting messaging frameworks that allow messages to be either synchronous or asynchronous. The base synchronous support in TransferQueues mirrors our fair synchronous queue. The asynchronous additions differ only by releasing producers before items are taken.

Although we have improved the scalability of the synchronous queue, there may remain potential for improvement in some contexts. Most of the inter-thread contention in enqueue and dequeue operations occurs at the memory containing the head (and, for fair queues, tail). Reducing such contention by spreading it out is the idea behind *elimination* techniques introduced by Shavit and Touitou.[20] These may be applied to components featuring pairs of operations that collectively effect no change to a data structure, for example, a concurrent push and pop on a stack. Using elimination, multiple locations (comprising an *arena*) are employed as potential targets of the main atomic instructions underlying these operations. If two threads meet in one of these lower-traffic areas, they cancel each other out. Otherwise, the threads must eventually fall back (usually, in a tree-like fashion) to try the main location.

Elimination techniques have been used by Hendler et al.[4] to improve the scalability of stacks, and by us[18] to improve the scalability of the swapping channels in the *java.util.concurrent Exchanger* class. Moir et al.[15] have also used elimination in concurrent queues, although at the price of weaker ordering semantics than desired in some applications due to stack-like (LIFO) operation of the elimination arena. Similar ideas could be applied to our synchronous queues. However, to be worthwhile here, the reduced contention benefits would need to outweigh the delayed release (lower throughput) experienced when threads do not meet in arena locations. In preliminary work, we have found elimination to be beneficial only in cases of artificially extreme contention. We leave fuller exploration to future work.

Acknowledgments

We are grateful to Dave Dice, Brian Goetz, David Holmes, Mark Moir, Bill Pugh, and the PPoPP referees for feedback that significantly improved the presentation of this paper. This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

References

1. Anderson, T.E., Lazowska, E.D., Levy, H.M. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1631–1644.
2. Andrews, G.R. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
3. Hanson, D.R. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1997.
4. Hendler, D., Shavit, N., Yerushalmi, L. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Jun. 2004), 206–215.
5. Herlihy, M. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan. 1991), 124–149.
6. Herlihy, M., Luchangco, V., Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (May 2003).
7. Herlihy, M.P., Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3 (Jul. 1990), 463–492.
8. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
9. Karlin, A.R., Li, K., Manasse, M.S., Owicki, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), 41–55.
10. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (Aug. 1974), 453–455.
11. Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 1–11.
12. Lea, D. The java.util.concurrent Synchronizer Framework. *Sci. Comput. Prog. 58*, 3 (Dec. 2005), 293–309.
13. Mellor-Crummey, J.M., Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 21–65.
14. Michael, M.M., Scott, M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing* (May 1996), 267–275.
15. Moir, M., Nussbaum, D., Shalev, O., Shavit, N. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Jul. 2005), 253–262.
16. Scherer III, W.N. Synchronization and concurrency in user-level software systems. Ph.D. dissertation, Department of Computer Science, University of Rochester (Jan. 2006).
17. Scherer III, W.N., Lea, D., Scott, M.L. Scalable synchronous queues. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming* (Mar. 2006).
18. Scherer III, W.N., Lea, D., Scott, M.L. A scalable elimination-based exchange channel. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages* (Oct. 2005). In conjunction with OOPSLA '05.
19. Scherer III, W.N., Scott, M.L. Nonblocking concurrent objects with condition synchronization. In *Proceedings of the 18th International Symposium on Distributed Computing* (Oct. 2004).
20. Shavit, N., Touitou, D. Elimination trees and the construction of pools and stacks. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures* (Jul. 1995).
21. Treiber, R.K. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

William N. Scherer III (scherer@edu)
Department of Computer Science
Rice University, Houston, TX.

Doug Lea (dl@cs.oswego.edu)
Department of Computer Science
SUNY Oswego, Oswego, NY.

Michael L. Scott (scott@cs.rochester.edu)
Department of Computer Science
University of Rochester, Rochester, NY.

Pacific Northwest National Laboratory
Software Architect

The Applied Computer Science Group within the Computational Sciences & Mathematics Division seeks an experienced software architect to lead R&D projects in a variety of scientific application domains. The ideal candidate will have extensive technical knowledge and successful demonstration of designing and delivering complex, middleware-based systems that can support a wide range of specific problem areas. They will also be capable of innovating to create next-generation architectures to solve data intensive computing problems, work with clients to propose and develop projects, and publish results in the scientific literature.

The Computational Sciences & Mathematics Division at PNNL provides science, technologies, and leadership to solve significant problems of national interest in energy, environment, national security, and fundamental science. Our research is aimed at creating new, state-of-the-art computational capabilities using extreme-scale simulation and peta-scale data analytics that enable scientific breakthroughs. Computational scientists at PNNL have interdisciplinary expertise in high-performance computing; data management research and development; extreme workflow management for scientific applications; mathematics; and computational biology and bioinformatics. We invite you to apply for this position and join a dynamic organization that is advancing scientific frontiers.

U.S. citizenship and the ability to obtain a U.S. Department of Energy clearance are required.

To apply or learn more about the responsibilities and minimum requirements to quality for the Software Architect, go to http://jobs.pnl.gov and post your resume to **Job ID 116621**. You can learn more about our organization at http://www.pnl.gov/science/.

Tulane University
Lecturer in Computational Science

The Tulane University Center for Computational Science seeks a Masters-level instructor starting fall 2009. Candidates must be excellent teachers for C++ Programming, Data Visualization & Large Scale Computing. Visit http://www.ccs.tulane.edu.

University of Massachusetts Lowell
Department of Computer Science
Faculty Position at All Ranks

Anticipated tenure-track faculty position to start in September 2009. Rank/tenure status dependent on qualifications.

UMass Lowell is located 30 miles northwest of Boston in the high-tech corridor of Massachusetts. Department has 18 tenured/ and tenure-track faculty and offers degree programs at bachelor's, master's, and doctoral levels. Also offers bioinformatics options at all levels; computational mathematics option at doctoral level.

Computer Science faculty received approximately $3.8M in last two years in external research funding from NSF, DoD, DOH, and corporations, including two NSF Career awards.

Must hold PhD in computer science or closely related discipline, and be committed to developing/sustaining externally funded research program. Preference to outstanding candidates in areas of data mining and databases, who would be able to teach one graduate-level core course, which include algorithms, computing theory, and design of programming languages.

Exceptional senior level candidates in any major computer science research area will be considered (those who have made substantial contributions to their fields and have ongoing research projects funded by major US funding agencies). In addition to developing a research program, will contribute to collaborative research of existing departmental groups.

More information at: http://www.cs.uml.edu

Review of applications will begin immediately, continuing until position is filled. Applications received by April 1, 2009 will receive full consideration. Women and under-represented minorities strongly encouraged to apply.

To Apply:
Mail current CV, research statement, teaching statement, selected relevant publications, and

residency status to address below. Applicants for Assistant Professor should also arrange three letters of recommendations sent directly.

Search Committee for Department of Computer Science
University of Massachusetts Lowell
One University Avenue
Lowell, MA 01854
Job Reference Number: FC04070901

Or

E-mail (preferred) all materials stated above to: hiring@cs.uml.edu.

Include reference number in subject line of e-mail. Applicants for Assistant Professor should also arrange three letters of recommendations sent directly.

The University of Massachusetts is an Equal Opportunity/Affirmative Action Title IX, H/V, ADA 1990 Employer and Executive Order 11246, 41 CFR60-741 4, 41 CRF60-250 4, 41CRF60-1 40 and 41 CFR60-1,4 are hereby incorporated. Please include reference number in subject line of e-mail.

Vrije Universiteit

Postdoc Positions Available in Amsterdam

The Department of Computer Science at the Vrije Universiteit is looking for two postdocs and a programmer to work in the group of Prof. Andrew Tanenbaum. Our research is about how to design and build dependable and secure systems software. For more information about the positions, please see www.cs.vu.nl/~ast/jobs

DOI:10.1145/1506409.1506432

Peter Winkler

Puzzled
Understanding Relationships Among Numbers

Welcome to three new challenging mathematical puzzles. Solutions to the first two will be published next month; the third is as yet (famously) unsolved. In each puzzle, the issue is how numbers interact with one another.

1. A colony of chameleons includes 20 red, 18 blue, and 16 green individuals. Whenever two chameleons of different colors meet, each changes to the third color. Some time passes during which no chameleons are born or die nor do any enter or leave the colony. Is it possible that at the end of this period, all 54 chameleons are the same color?

2. Four non-negative integers are written on a line. Below each number, now write the (absolute) difference between that number and the one to its right (that is, the result of subtracting the smaller from the larger of the two numbers). Below the fourth, write the absolute difference between it and the first number. The result is a new row of four non-negative integers. These four subtractions constitute one "operation" you can repeat on the four new numbers. Now show that after a finite number of such operations, you must reach a point where all four numbers are 0.

For example, if you start with the sequence 43, 11, 21, 3, here's what happens:

43	11	21	3
32	10	18	40
22	8	22	8
14	14	14	14
0	0	0	0

As you see, 0 0 0 0 was reached after only four operations.

Try it yourself with, say, random numbers between 0 and 100; you'll be amazed how quickly you get to 0 0 0 0.

Note, however, that if you do the same thing with five numbers, you might never stop. If you found the first part of this problem too easy, try answering this question: For which n is it the case that this process, beginning with n numbers, always gets you to n zeroes?

3. The "lonely runner," an intriguing open problem in number theory, asks whether the following is true: Suppose you are one of n runners who start together on a circular track one kilometer in length, each running at a different constant speed. Then, at some moment in time you will be at distance at least $1/n$ kilometers from all the other runners. Note when the ratios between speeds are irrational, as they would, almost surely, be, if the speeds were, say, random real numbers between 0 and 1, then it is indeed true. It's when the speeds are rationally related that things start to get interesting.

Readers are encouraged to submit prospective puzzles for future columns to puzzled@cacm.acm.org.

Peter Winkler (puzzled@cacm.acm.org) is Professor of Mathematics and of Computer Science and Albert Bradley Third Century Professor in the Sciences at Dartmouth College, Hanover, NH.

ACM Digital Library

www.acm.org/dl

The Ultimate Online INFORMATION TECHNOLOGY Resource!

Powerful and vast in scope, the **ACM Digital Library** is the ultimate online resource offering unlimited access and value!

The **ACM Digital Library** interface includes:

- **The ACM Digital Library** offers over 40 publications including all ACM journals, magazines, and conference proceedings, plus vast archives, representing over 2 million pages of text. The ACM DL includes full-text articles from all ACM publications dating back to the 1950s, as well as third-party content with selected archives. PLUS NEW: Author Profile Pages with citation and usage counts and New Guided Navigation search functionality! **www.acm.org/dl**

- **The Guide to Computing Literature** offers an enormous bank of over one million bibliographic citations extending far beyond ACM's proprietary literature, covering all types of works in computing such as journals, proceedings, books, technical reports, and theses! **www.acm.org/guide**

- **The Online Computing Reviews Service** includes reviews by computing experts, providing timely commentary and critiques of the most essential books and articles.

Available only to ACM Members.
Join ACM online at **www.acm.org/joinacm**

To join ACM and/or subscribe to the Digital Library, contact ACM:

Phone: 1.800.342.6626 (U.S. and Canada)
+1.212.626.0500 (Global)
Fax: +1.212.944.1318
Hours: 8:30 a.m.-4:30 p.m., Eastern Time

Email: acmhelp@acm.org
Join URL: www.acm.org/joinacm
Mail: ACM Member Services
General Post Office
PO Box 30777
New York, NY 10087-0777 USA

*Guide access is included with Professional, Student and SIG membership. ACM Professional Members can add the full ACM Digital Library for only $99 (USD). Student Portal Package membership includes the Digital Library. Institutional, Corporate, and Consortia Packages are also available.

Association for Computing Machinery

Advancing Computing as a Science & Profession

AD29

The Best Place to Find the Perfect Job... Is Just a Click Away!

No need to get lost on commercial job boards.
The ACM Career & Job Center is tailored specifically for you.

JOBSEEKERS

- Manage your job search
- Access hundreds of corporate job postings
- Post an anonymous resume
- Advanced Job Alert system

EMPLOYERS

- Quickly post job openings
- Manage your online recruiting efforts
- Advanced resume searching capabilities
- Reach targeted & qualified candidates

NEVER LET A JOB OPPORTUNITY PASS YOU BY!

START YOUR JOB SEARCH TODAY!

http://www.acm.org/careercenter

Association for Computing Machinery

Advancing Computing as a Science & Profession

POWERED BY JOBTARGET