

KV the Konqueror

Vol. 3 No. 7

acm
queue
architecting tomorrow's computing

September 2005

Debating the
Fuzzy Boundaries

Make Way for **Multiprocessors**

Scaling with CMPs

Performance at What Price?

Multicore Programming

**JUNK MAIL.
MISSING THE TRAIN.
APPLICATION
INTEGRATION.
HAIR LOSS.
BAD MUSIC.
PAPER CUTS.**

.....
**WINDOWS SERVER SYSTEM
WITH .NET MAKES APPLICATION
INTEGRATION LESS PAINFUL.**
.....

The list of potentially unpleasant things in your day is long. But now, the list just got a bit shorter. How? With the help of Windows Server System™ and Microsoft® .NET.

The .NET Framework, an integral component of Windows Server System, is the development and execution environment that allows your different components and applications to work together seamlessly. That means applications are easier to build, manage, deploy,

and integrate. The .NET Framework uses industry standards such as XML and Web Services which allow enterprise applications to be connected to infrastructure of any kind. In addition, the productivity enhancing features of .NET, such as automatic mapping of data to and from XML, also help to simplify integration by reducing the amount of code required to get it all done.

Find out more about application integration with Windows Server System and .NET: Simply get the Connected Systems Resource Kit at microsoft.com/connectedsystems



WHEN THINGS GET UNPLEASANT, JUST RELAX AND
THINK OF EASIER APPLICATION INTEGRATION.

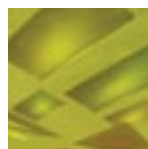
CONTENTS

SEPTEMBER 2005

VOL. 3 NO. 7



MULTIPROCESSORS



The Future of Microprocessors 26

Kunle Olukotun and Lance Hammond, Stanford University
The transition to chip multiprocessors is inevitable.
Are you prepared to leverage their power?



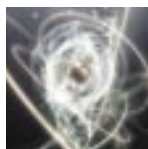
Extreme Software Scaling 36

Richard McDougall, Sun Microsystems
Scaling with multiprocessors is no longer
just for boutique, high-end servers.



The Price of Performance 48

Luiz André Barroso, Google
The question is not, "Can you afford to switch to
multicore CPUs?" but rather, "Can you afford *not* to?"



Software and the Concurrency Revolution 54

Herb Sutter and James Larus, Microsoft
Multicore hardware architectures are blazing
into the future. What's the next step for
the software community?

Innovations by InterSystems



Rapid development with robust objects



Lightning speed with a multidimensional engine



Easy database administration



Massive scalability on minimal hardware

Database With Multidimensional Appeal.

Caché is the first multidimensional database for transaction processing and real-time analytics. Its post-relational technology combines robust objects and robust SQL, thus eliminating object-relational mapping. It delivers massive scalability on minimal hardware, requires little administration, and incorporates a rapid application development environment.

These innovations mean faster time-to-market, lower cost of operations, and higher application performance. We back these claims with this money-back guarantee: *Buy Caché for new application development, and for up to one year you can return the license for a full refund if you are unhappy for any reason.** Caché is available for Unix, Linux, Windows, Mac OS X, and OpenVMS – and it's deployed on more than 100,000 systems ranging from two to over 50,000 users. We are InterSystems, a global software company with a track record of innovation for more than 25 years.



Try an innovative database for free: Download a fully functional, non-expiring copy of Caché, or request it on CD, at www.InterSystems.com/Cache15S

* Read about our money-back guarantee at the web page shown above.

© 2005 InterSystems Corporation. All rights reserved. InterSystems Caché is a registered trademark of InterSystems Corporation. 8-05 CacheInno15Queue



CONTENTS

INTERVIEW



A CONVERSATION WITH

ROGER SESSIONS AND TERRY COATTA 16
Queue board member Terry Coatta and
“Fuzzy Boundaries” author Roger Sessions
spar on the differences between objects,
components, and Web services.

DEPARTMENTS

NEWS 2.0 8

Taking a second look at the news
so that you don’t have to.

WHAT’S ON YOUR HARD DRIVE? 10

Visitors to our Web site are invited to tell us about
the tools they love—and the tools they hate.

KODE VICIOUS 12

KV the Konqueror
George V. Neville-Neil, Consultant

CURMUDGEON 64

Multicore CPUs for the Masses
Mache Creeger, Emergent Technology Associates

Know where you are,
Know where you're going,
Know where you've been

with DevTest.



DevTest

DevTest manages your test coverage smarter giving you a complete view of what you've tested and what you still need to test without having to break out the spreadsheets.

- Create and manage test definitions that can be shared across environments and teams
- Establish testing targets for an entire release and then let DevTest compare your estimates to the actual results
- Schedule and assign tasks via the DevTest planning wizard
- Execute tasks, search for existing defects and submit new defects in DevTrack without having to leave DevTest
- Associate test tasks with defects or change requests and track each test task's history
- Analyze test results with built-in presentation quality reports

TechExcel

www.techexcel.com | 1-800-439-7782

Publisher and Editor

Charlene O'Hanlon
cohanlon@acmqueue.com

Editorial Staff

Executive Editor

Jim Maurer
jmaurer@acmqueue.com

Associate Managing Editor

John Stanik
jstanik@acmqueue.com

Copy Editor

Susan Holly

Art Director

Sharon Reuter

Production Manager

Lynn D'Addesio-Kraus

Copyright

Deborah Cotton

Editorial Advisory Board

Eric Allman
Charles Beeler
Steve Bourne
David J. Brown
Terry Coatta
Mark Compton
Stu Feldman
Ben Fried
Jim Gray
Randy Harr
Wendy Kellogg
Marshall Kirk McKusick
George Neville-Neil

Sales Staff

National Sales Director

Ginny Pohlman
415-383-0203
gpohlman@acmqueue.com

Regional Eastern Manager

Walter Andrzejewski
207-763-4772
walter@acmqueue.com

Regional Midwestern/ Southern Manager

Sal Alioto
843-236-8823
salalioto@acmqueue.com

Contact Points

Queue editorial
queue-ed@acm.org

Queue advertising
queue-ads@acm.org

Copyright permissions
permissions@acm.org

Queue subscriptions
orders@acm.org

Change of address
acmcoa@acm.org

ACM Headquarters

Executive Director and CEO: John White
Director, ACM U.S. Public Policy Office: Jeff Grove

Deputy Executive Director and COO: Patricia Ryan
Director, Office of Information Systems: Wayne Graves
Director, Financial Operations Planning: Russell Harris
Director, Office of Membership: Lillian Israel

Director, Office of Publications: Mark Mandelbaum
Deputy Director, Electronic Publishing: Bernard Rous
Deputy Director, Magazine Development: Diane Crawford
Publisher, ACM Books and Journals: Jono Hardjowirogo

Director, Office of SIG Services: Donna Baglio
Assistant Director, Office of SIG Services: Erica Johnson

Executive Committee

President: Dave Patterson
Vice-President: Stuart Feldman
Secretary/Treasurer: Laura Hill
Past President: Maria Klawe
Chair, SIG Board: Robert A. Walker

For information from Headquarters: (212) 869-7440

ACM U.S. Public Policy Office: Cameron Wilson, Director
1100 17th Street, NW, Suite 507, Washington, DC 20036 USA
+1-202-659-9711-office, +1-202-667-1066-fax, wilson_c@acm.org

ACM Copyright Notice: Copyright © 2005 by Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. Copyright for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to republish from: Publications Dept. ACM, Inc. Fax +1 (212) 869-0481 or e-mail <permissions@acm.org>

For other copying of articles that carry a code at the bottom of the first or last page or screen display, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, 508-750-8500, 508-750-4470 (fax).



ACM Queue (ISSN 1542-7730) is published ten times per year by the ACM, 1515 Broadway, New York, NY, 10036-5701. POSTMASTER: Please send address changes to ACM Queue, 1515 Broadway, New York, NY 10036-5701 USA. Printed in the U.S.A.

The opinions expressed by ACM Queue authors are their own, and are not necessarily those of ACM or ACM Queue.

Subscription information available online at www.acmqueue.com.
BPA Worldwide Membership applied for October 2004

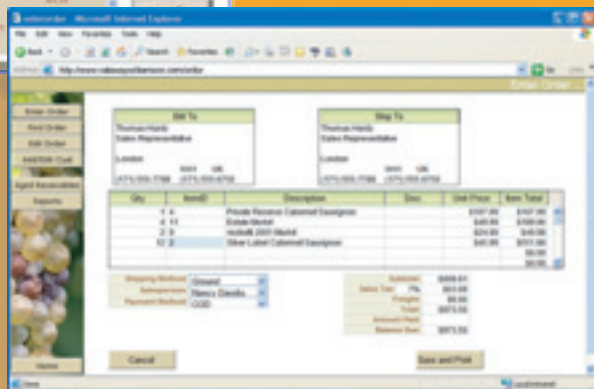


Would you believe that each component within this application is a FarPoint Spread?

FarPoint Spread bound only by the limits of your creativity!

"Spread made it possible to deliver an intuitive application that allows management to forecast sales trends in a fraction of the time it normally would have taken to complete. Keep up the great work!"

Larry Smith
Sr. Analyst/Programmer
Gerber Products Company -
Novartis Consumer Health N.A.



Handling data involves more than simply typing numbers in a spreadsheet. That's why today's application developers use FarPoint Spread, a powerful component that gives you control over the complexity in .NET or COM applications for advanced user interaction and data manipulation. FarPoint is no longer simply refining spreadsheet development with this component, but redefining component architecture to deliver a powerful programming experience in a small package. With cell-level control, databinding capabilities, over

300 formula functions, and drawing shapes, Spread helps you do more than simply create spreadsheet applications. With its underlying object-oriented model, you can extend the powerful API and customize the appearance and interaction of the interface for a range of uses. From user-defined formula functions, to the hierarchical data support that permits binding to relational data, to the default cell type that handles user entry based on logic you define, you can depend on FarPoint Spread to satisfy your most demanding requirements.

FarPoint Spread

refining spreadsheet development...
redefining component architecture

With a component like this, application development can be creative and fun, saving you time and effort while still giving you the functionality you need. And all of that power doesn't come at the expense of ease of use; Spread's visual designers expose much of the feature set at design-time.

Download a trial version today from www.fpoint.com and see for yourself how fun application development can be when you develop with Spread!

For more information:

www.fpoint.com

Telephone: 919-460-4551
1-800-645-5913

"Spread has allowed us to develop a key enterprise wide business application in only 11 months. We used Spread for the navigation form and found it to be an effective way to present a 3 level hierarchy to a user."

Pete Cormier
Vice President & Manager IT - ALM Services, Inc.

FarPoint



Open Source/2

IBM recently announced that it would discontinue support for its once-flagship operating system, OS/2, beginning in late 2006. Developed in the 1980s during an early alliance with Microsoft, OS/2 eventually became OS/2 Warp and had some success during the '90s, particularly in the server market. But its desktop counterpart failed to take off, and IBM eventually ceded victory to Microsoft. IBM is now urging OS/2 users to switch to Linux, which it supports. Switch to Linux? If only it were that easy. Though gone from the spotlight, OS/2 continues to run on servers around the globe, especially on those linked to ATMs. Accordingly, there remains an active community of OS/2 users, many of whom believe that OS/2 is superior to more popular alternatives in some areas (e.g., security, file system). Emblematic of this support is a petition recently signed by nearly 10,000 OS/2 users, urging IBM to make OS/2 open source.

The problem? In addition to the fact that IBM initially co-developed OS/2 with Microsoft, the operating system contains thousands of lines of code owned by third parties, so unraveling the intellectual property rights would be daunting. But loyal OS/2 users feel that, if nothing else, releasing even portions of the code would yield a useful educational resource. Whether that means learning what *to* do or what *not* to do when building an operating system is open for debate.

WANT MORE?

<http://news.zdnet.co.uk/0,39020330,39209811,00.htm>

Anti-spam Activism ... or Vigilantism?

By now it's clear that current legislation enacted to crack down on spam is ineffective by itself. We also need sound *technological* solutions to the spam problem. Much work is being done on this front. Spam-filtering tools have become ubiquitous, and promising new innovations such as SMTP Path Analysis, which uses IP information in the message header to determine the legitimacy of e-mail messages, are expanding our arsenal in the anti-spam war.

But for those who believe in taking more drastic measures, there is Blue Frog. Currently a free anti-spam solution offered by Blue Security, Blue Frog works by inviting users to add their e-mail addresses to a "do not spam" list. For each person added to the list, several fake

Taking a second look AT THE NEWS SO YOU DON'T HAVE TO

e-mail addresses are created, resulting in a "honey pot" that lures spammers. Spammers who send messages to those addresses are first warned to cease doing

so. If the warnings are ignored, the software triggers each user on the list to send a complaint to the URL contained in the spam. Thousands of simultaneous complaints will cripple the spammer's Web server. Honest community activism? Illegal denial of service? We'll let you decide. Slippery terrain, indeed.

WANT MORE?

<http://www.linuxinsider.com/story/44867.html>

Ride, Robot, Ride

Don't say you didn't see it coming. The latest generation of robotic technology has finally arrived: robotic camel jockeys. Oh, you're not from the United Arab Emirates? Well, let us fill you in. Camel racing, an ancient and, according to one UAE official, "indispensable" spectator sport (i.e., lots of wagering), has long been met with derision by human rights activists who criticize the sport for allowing young children to participate. They further allege that the child camel jockeys, sometimes as young as 4 years old, have been kidnapped and deliberately starved to make them as lean and mean as possible.

An answer to the critics came from a Swiss company contracted to build humanoid robots that are set to take the place of their imperiled child predecessors. The robots "sit" near the rear of the camel (post-hump) and balance with short, mechanical legs. They hold the reins with mechanical arms and hands. What might disappoint robotics enthusiasts is that these robot jockeys are not entirely autonomous; they are operated from the sidelines via remote control. This is just the beginning, though, and who knows whether more autonomous models eventually will make their way onto the sandy tracks.

No comment yet from the U.S. horse racing community, whose jockeys have been similarly criticized for having to endure grueling privations to make weight. Churchill Downs, look out!

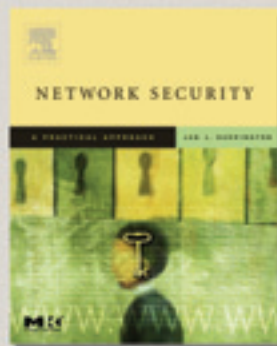
WANT MORE?

<http://www.newscientist.com/article.ns?id=dn7705> Q



MK
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
www.mkp.com

Community ♦ Quality ♦ Excellence



NETWORK SECURITY:
A Practical Approach
by Jan L. Harrington



JOE CELKO'S SQL FOR SMARTIES: Advanced SQL Programming, 3rd Edition
by Joe Celko



GRID COMPUTING:
The Savvy Manager's Guide
by Pawel Plaszczak and Richard Wellner, Jr.



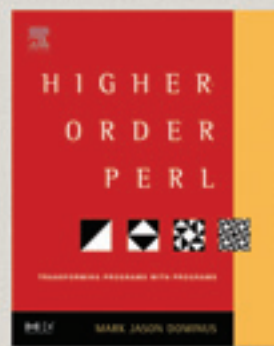
MOVING OBJECTS DATABASES
by Ralf Hartmut Güting and Markus Schneider



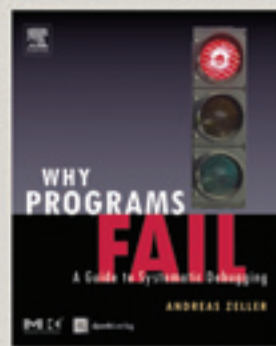
FUZZY MODELING AND GENETIC ALGORITHMS FOR DATA MINING AND EXPLORATION
by Earl Cox



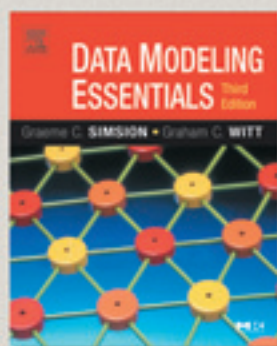
INNOVATION HAPPENS ELSEWHERE: Open Source as Business Strategy
by Ron Goldman and Richard P. Gabriel



HIGHER ORDER PERL: Transforming Programs with Programs
by Mark Jason Dominus



WHY PROGRAMS FAIL: A Guide to Systematic Debugging
by Andreas Zeller



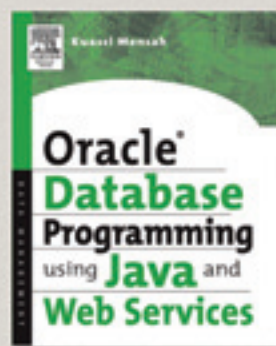
DATA MODELING ESSENTIALS, THIRD EDITION
by Graeme Simsion and Graham C. Witt



INSTANT MESSAGING SECURITY
by John W. Rittinghouse and James F. Ransome



DEPLOYING LINUX ON THE DESKTOP
by Edward L. Halletky



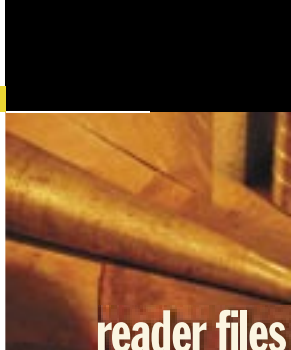
ORACLE DATABASE PROGRAMMING USING JAVA AND WEB SERVICES
by Kuassi Mensah

ORDER FROM MORGAN KAUFMANN AND RECEIVE A 15% DISCOUNT WITH FREE SHIPPING!
Refer to offer code: 82366

Mail: Elsevier, Order Fulfillment Department, 11830 Westline Drive, St. Louis, MO 63146

Phone: (800) 545-2522 / (314) 453-7010 [Intl.] **Fax:** (800) 535-9935 / (314) 453-7095 [Intl.]

Web: www.mkp.com **Email:** usbkinfo@elsevier.com **Volume Discounts:** naspecialsales@elsevier.com



What's on Your Hard Drive?

WOYHD is a forum for expressing your opinions on the tools you love and loathe. Tools, as we've made clear, can be anything from programming languages to IDEs to database products. This month we've taken liberties with the definition to include the Linux music software praised below. Listening to music can help

us get through those long hours spent unraveling lines of spaghetti code (someone else's, of course). It can also make some people completely unproductive, but cannot the same be said of many other, more obvious "tools"? Don't get us wrong, though—never, under any circumstances, will Minesweeper be considered a tool!

Who: Chris Bellini

What industry: Manufacturing

Job title: Software developer

Flavor: Develops on Windows for Windows



Tool I love! Python. I'm still a newbie to Python but I'm quite impressed with it thus far. As a scripting language, it can quickly test an idea or an algorithm, even if the project I'm working on doesn't

use Python. Also, with free tools such as wxPython and py2exe, a Python script can easily become a full-blown distributable application with a robust UI.

Tool I hate! Microsoft Visual Studio .NET. It's a love/hate relationship. On the one hand, it's my bread and butter and I've learned to use many of its features. On the other hand, it has become a bloated resource hog that makes things sluggish while a large app, such as CAD/CAM, is running simultaneously.



Who: John Styles

What industry: Technology vendor

Job title: Chief architect

Flavor: Develops on Windows for Windows



Tool I love! Awk. Within its chosen problem domain, nothing can beat Awk's elegance and simplicity. It is easy to deploy, and its syntax is simpler and more rational than certain other scripting languages I could mention. I often try other tools but come back to the one that always does the job.

Tool I hate! OLE DB. If OLE DB were just another bizarre, overly complex, overly abstract, poorly documented API, then I could just happily ignore it. Unfortunately, it is the native data access API for Microsoft SQL Server, so it cannot be avoided. If only there were a decent simple C language API for it, such as Oracle's OCI.



Who: Guilherme Mauro Germoglio Barbosa

What industry: Education

Job title: Software developer

Flavor: Develops on Linux for Linux



Tool I love! XMMS. I simply cannot focus on my work without any music. Programming is a bit of a lonely task, and listening to music helps combat this loneliness. 24/7 listening to music = 24/7 programming!

Tool I hate! CTTE. This tool is a bit buggy—only five minutes of use produced six exceptions. I hope they improve it. Maybe it will be very useful in the future. But there's another problem: programming is fun—crafting artifacts that no one really cares about (such as those created by CTTE) is not.



Who: Jeff Price

What industry: Not-for-profit

Job title: Software engineer

Flavor: Develops on Windows for Unix



Tool I love! Eclipse. Eclipse allows me to be much more productive by assisting with syntax and generating many standard, repetitive code blocks. The refactoring tools make otherwise unthinkable-to-tackle tasks (such as renaming/repackaging a frequently used class) almost trivial.

Tool I hate! PVCS Version Manager. The X interface is slow, clunky, and unstable. Open projects sometimes disappear, many error messages inaccurately represent the cause of the problem...and did I mention that it's slow? It represents the antithesis of the productivity gains I get by using Eclipse.





GET YOUR STAFF TO WRITE BETTER CODE, FASTER.

IF THAT MAKES YOU GIGGLE WITH EXCITEMENT,
YOU AIN'T SEEN NOTHIN' YET.

BIGGER BRAINS = BIGGER BOTTOM LINE. TURN YOUR TEAM INTO WINTELLECTUALS TODAY. FOR SUPERIOR .NET CONSULTING, TRAINING, DEBUGGING, CALL 877.968.5528 OR VISIT WIntellect.com

Wintellect®
Know how.



KV the Konqueror

It's been a couple of months, and Kode Vicious has finally returned from his summer vacation. We asked him about his travels and the only response we got was this: "The South Pole during winter ain't all it's cracked up to be!" Fortunately, he made it back in one piece and is embracing the (Northern hemisphere's) late summer balminess with a fresh installment of koding kwestions. This month, KV follows up on a security question from a previous column and then revisits one of koding's most divisive issues: language choice. Welcome back!

Dear KV,

Suppose I'm a customer of Sincere-and-Authentic's ("Kode Vicious Battles On," April 2005:15-17), and suppose the sysadmin at my ISP is an unscrupulous, albeit music-loving, geek. He figured out that I have an account with Sincere-and-Authentic. He put in a filter in the access router to log all packets belonging to a session between me and S&A. He would later mine the logs and retrieve the music—without paying for it.

I know this is a far-fetched scenario, but if S&A wants his business secured as watertight as possible, shouldn't he be contemplating addressing it, too? Yes, of course, S&A will have to weigh the risk against the cost of mitigating it, and he may well decide to live with the risk. But I think your correspondent's suggestion is at least worthy of a summary debate—not something that should draw disgusted looks!

There is, in fact, another advantage to encrypting the payload, assuming that IPsec (Internet Protocol security) isn't being used: decryption will require special clients, and that will protect S&A that much more against the theft of merchandise.

Balancing is the Best Defense

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

Dear Balancing,

Thank you for reading my column in the April 2005 issue of *Queue*. It's nice to know that someone is

paying attention. Of course, if you had been paying closer attention, you would have noticed that S&A said, "In the design meeting about this I suggested we just encrypt all the connections from the users to the Web service because that would provide the most protection for them and us." That phrase, "just encrypt all the connections," is where the problem lies.

Your scenario is not so far-fetched, but S&A's suggestion of encrypting all the connections would not address the problem. Once users have gotten the music without their evil ISP's sniffing it, they would still be able to redistribute the music themselves. Or, the evil network admin would sign up for the service and simply split the cost with, say, 10 of his music-loving friends, thereby getting the goods at a hefty discount. What S&A really needs is what is now called digital rights management. It's called this because for some reason we let the lawyers and the marketing people into the industry instead of doing with them what was suggested in Shakespeare's *Henry VI*.

What S&A failed to realize was that the biggest risk of revenue loss was not in the network, where only a small percentage of people can play tricks as your ISP network administrator can, but at the distribution and reception points of the music. Someone who works for you walking off with your valuable information is far more likely than someone trying to sniff packets from the network. Since computers can make perfect copies of data (after all, that's how we designed these things in the first place), it is the data itself that must be protected, from one end of the system to the other, in order to keep from losing revenue.

All too often, people do not consider the end-to-end design of their systems and instead try to fix just one part.

KV

Dear KV,

Since there was some debate in my company over the following issue, I'm curious to see what you believe: put-

Perforce.

The *fast* SCM system.

For developers who don't like to wait.



Tired of using a software configuration management system that stops you from checking in your files? Perforce SCM is different: fast and powerful, elegant and clean. Perforce works at your speed.

[Fast]

[Scalable]

[Distributed]

Perforce's lock on performance rests firmly on three pillars of design. A carefully keyed relational database ensures a rapid response time for small operations plus high throughput when the requests get big - millions of files big. An efficient streaming network protocol minimizes the effects of latency and maximizes the benefits of bandwidth. And an intelligent, server-centric data model keeps both the database and network performing at top speed.

It's your call. Do you want to work, or do you want to wait?



Download a free copy of Perforce, no questions asked, from www.perforce.com. Free technical support is available throughout your evaluation.

All trademarks used herein are either the trademarks or registered trademarks of their respective owners.

ting aside performance issues (which I think are relatively minor on modern PCs), when would you recommend using C++ for development, and when would you recommend C? Do you think it is always better to use C++?

My feeling is that unless your application is inherently object oriented (e.g., user interfaces), C++ will tend to make the implementation worse instead of making it better (e.g., constructors and operators doing funny unexpected things; C++ experts trying to “use their expertise” and writing C++ code that is very efficient but extremely hard to read and not portable; huge portability—and performance—issues when using templates; incomprehensible compiler/linker error messages; etc., etc.). I also think that although people can write bad C code (gotos out of macros was a nice one), typically people can write *awful* C++ code. Where do you stand on this dispute?

Wondering How Much + There is in ++

Dear Wondering,

Choosing a language is something I’ve addressed before in other letters, but the C vs. C++ debate has raged as long as the two languages have been in existence, and, really, it’s getting a bit tiring. I mean, we all know that assembler is the language that all red-blooded programmers use! Oh, no, wait, that’s not it.

I’m glad you ask this question, though, because it gives me license to rant about it—and to dispel a few myths.

The first, and most obvious, myth in your letter is that user interfaces are inherently object oriented. Although many introductory textbooks on object-oriented programming have user interfaces as their examples, this has a lot more to do with the fact that humans like pretty pictures. It is far easier to make a point graphically than with text. I have worked on object-oriented device drivers, which are about as far as you’ll ever get from a user interface.

Another myth that your letter could promulgate is that C is not an object-oriented language. A good example of object-oriented software in C is the vnode filesystem interface in BSD Unix and other operating systems. So, if you want to write a piece of object-oriented software, you can certainly do it in C or C++, or assembler for that matter.

One final myth, which was actually dispelled by Donn Seeley in “How Not To Write Fortran in Any Language” (*ACM Queue*, December/January 2004-2005:58-65), is that C++ leads to less understandable code than C. Over the past 20 years I have seen C code that was spaghetti and C++ code that was a joy to work on, and vice versa.

So, after all that myth bashing, what are we left with?

Well, the things that are truly important in picking a language are:

- What language is most of the team experienced in? If you’re working with a team and six out of eight of them are well versed in C but only two know C++, then you’re putting your project, and job, at risk in picking C++. Perhaps the two C++ koders can teach the C folks enough C++ to be effective but it’s unlikely. To estimate the amount of work necessary for a task, you have to understand your tools. If you don’t normally use a nail gun, then you’re likely to take someone’s toe off with it. Losing toes is bad, as you need them for balance.
- Does the application require any of the features of the language you’re using? C and C++ are a lot alike as languages (i.e., in syntax), but they have different libraries of functions and different ways of working that may or may not be relevant to your application. Often realtime constraints require the use of C because of the control it can provide over the data types. If type safety is of paramount importance, then C++ is a better choice because that is a native part of the language that is not present in C.
- Does the application require services from other applications or libraries that are hard to use or debug from one or the other language? Creating shim layers between your code and the libraries you depend on is just another way of adding useless, and probably buggy, code to your system. Shim layers should be avoided like in-laws. They’re OK to talk about, and you might consider keeping them around for a week, but after that, out they go as so much excess, noisy baggage.

There are lots of other reasons to choose one language over another, but I suspect that the three listed here should be enough for you and your team to come to some agreement. You’ll notice that none of them has to do with how easy it is to understand templates or how hard it is to debug with exceptions.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor’s degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

© 2005 ACM 1542-7730/05/0900 \$5.00

igrep.com

Developer Search Engine?

COOL.





A Conversation with Roger Sessions and Terry Coatta

In the December/January 2004-2005 issue of *Queue*, Roger Sessions set off some fireworks with his article about objects, components, and Web services and which should be used when ("Fuzzy Boundaries," 40-47). Sessions is on the board of directors of the International Association of Software Architects, the author of six books, writes the Architect Technology Advisory, and is CEO of ObjectWatch. He has a very object-oriented viewpoint, not necessarily shared by *Queue* editorial board member Terry Coatta, who disagreed with much of what Sessions had to say in his article. Coatta is an active developer who has worked extensively with component frame-

The difference

BETWEEN OBJECTS AND
COMPONENTS?
THAT'S DEBATABLE.

works. He is vice president of products and strategy at Silicon Chalk, a startup software company in

Vancouver, British Colum-

bia. Silicon Chalk makes extensive use of Microsoft COM for building its application. Coatta previously worked at Open Text, where he architected CORBA-based infrastructures to support the company's enterprise products.

We decided to let these two battle it out in a forum that might prove useful to all of our readers. We enlisted another *Queue* editorial board member, Eric Allman, CTO



PHOTOGRAPH BY TOM UPTON



PHOTOGRAPH BY GEORGE BRAINARD



eclipse

Application Development Lifecycle

A special advertising supplement with this issue of Queue

Agitar™
SOFTWARE

DataMirror®
PointBase®

Exadel™
Extend. Adapt. Deliver.

my
eclipse

™ NitroX

PARASOFT®
We make software work.™

PERFORCE
SOFTWARE

SAP®

serena™

slickedit.

Telelogic

VERSANT

WIND RIVER

of Sendmail Inc., to moderate what we expected to be quite a provocative discussion. Our expectations were dead on.

ERIC ALLMAN I've talked to people who work on object-oriented stuff, who have read your "Fuzzy Boundaries" article, Roger, and every single one of them starts off by disagreeing that the difference between objects, components, and Web services is location-based.

Many of them speak of object-oriented RPCs (remote procedure calls), which aren't quite components. They are components that live together in a process and so forth. Since that was the fundamental point of your article, could you comment?

ROGER SESSIONS Unfortunately, none of these terms is very well defined. We're all using the terms as they make sense to us. Some of our disagreement may be simply semantic.

The component industry started with CORBA. The developers of CORBA were trying to solve one problem: distribution. They weren't trying to get objects to work together within the same process. Yes, you *could* have CORBA objects live together on the same machine, even in the same process, but that was not the main problem that CORBA cared about solving.

As far as Web services go, we could say that, yes, Web services *could* be in the same process or on the same machine. They *could* be in the same environment. But what was the essential problem Web services were trying to solve? It is about heterogeneous environments. It is about getting a .NET system to work with a WebSphere system, for example, not getting a .NET system to work with another .NET system.

TERRY COATTA It strikes me that it's hard to distinguish Web services from CORBA from EJB using that kind of rationale, because all three systems have open or at least standardized and available protocols. I can certainly make my WebSphere interoperate with an appropriate CORBA implementation that has the mappings for doing EJB. I can cross technology boundaries with all kinds of different standards.

RS If you're using the J2EE standards such as RMI (remote method invocation) over IIOP (Internet Inter-ORB Protocol), you are primarily going to be doing that within a single vendor's system, such as a WebSphere system. If you're going from a WebSphere system to a WebLogic system, your best shot at interoperability is through Web services. Why? Because you're crossing a technology boundary.

TC You're claiming that RMI over IIOP doesn't actually work?

RS It doesn't work for interoperability across technology boundaries.

TC There seem to be people out there getting it to work. Certainly, back in the days when I worked with CORBA there was no problem having different vendors' ORBs (object request brokers) interoperate with one another.



PHOTOGRAPH BY GEORGE BRAINARD

We used three or four of them at Open Text and had no difficulty at all with those environments interoperating with one another.

RS As long as you're going CORBA to CORBA, it works fine. But not when you are trying to get a CORBA system to work with a non-CORBA system.

TC But going from WebSphere to one of the other EJB vendors (e.g., WebLogic) in the CORBA space, there were probably five or six different major ORB vendors floating around, not to mention a couple of open source efforts, and all of those interoperated really well with one another.



RS CORBA to CORBA. They're all running on the same basic core of CORBA technology. The difference between that and Web services is that for Web services, unlike CORBA, there is no assumption whatsoever about what the underlying technology is.

TC That's not true. There's an assumption that one is using a certain set of protocols; otherwise, it doesn't work, and I mean CORBA was the same thing—a standard set of protocols. Nobody said that you had to actually implement the server-side aspect of the CORBA stuff to interoperate over the Internet. Everybody did because that's the way they defined the standards.

RS You could say the same thing about DCOM or RMI. While all of them support communications protocols, they, like CORBA, are about much more than communications protocols. They are about a platform. CORBA was 95 percent API, 5 percent interoperability. Web services is zero API and 100 percent interoperability.

TC That part I agree with, absolutely. That was probably the downfall of CORBA.

RS It's exactly the downfall of CORBA, and it will also be the downfall of J2EE. They didn't learn from that mistake.

EA Isn't Web services just essentially another standard for how to interact? The world has settled on CORBA protocols, not CORBA implementations. Wouldn't it have had exactly the same effect and maybe even better had the world agreed to use only the CORBA protocols?

RS It's quite possible, but the world didn't. CORBA lacked focus. The Web services effort has a lot of focus beyond interoperability.

The big difference between Web services and CORBA is that the Web services people said right from the beginning: there is no API. The only thing that we standardize is how messages go from one system to another and the coordination around that. CORBA was 95 percent about how the client binds into the system. That was its downfall.

TC Of course, from the perspective of a programmer, that's not necessarily a downfall, but a shortcoming. CORBA provided very nice interceptor architecture, a basic mechanism for dispatch, which everybody in Web services land has to rebuild from scratch. You can see that coming out now in the various Web services standards.

We were able to build an OTS (object transaction service) implementation on top of CORBA because of the appropriate interceptor mechanisms, support for global thread IDs, etc., etc. That work is taking a huge amount of time in Web services land, of course, because nobody has the infrastructure for it.

RS I've dedicated quite a few years of my life to CORBA,

and there were some very good ideas in it. Unfortunately, there was so much baggage that those good ideas were never allowed to flourish.

Hopefully we've learned from those mistakes. The only successful part of CORBA—of that massive effort, of those millions and millions and millions of dollars that were spent—was the tiny sliver of it that had to do with interoperability.

TC It wasn't just the interoperability. That was a big part of it, but the notion of a standard mechanism for interception and dispatch on the actual implementation side was also hugely successful because it allowed one to deploy things like OTS in a reasonable way without everybody having to basically rediscover from the ground up how to do that kind of stuff.

RS The reality is that CORBA is mostly about APIs, none of which anybody uses.

TC I agree. I was involved in the CORBA world, too, and of all of the interface specifications and the verticals, very little of them amounted to anything. But I think that although it's true historically to say one of the driving things behind CORBA was this desire to make things talk across the network to one another in interoperable fashion, the reality of it is that when people started using CORBA, they discovered the power that the standardized infrastructure offered. The basic server-side architecture, with standards for the dispatch mechanism, the interceptor mechanism, object lifecycle, and object identification, is an extremely powerful tool in the hands of developers actually delivering working systems.

RS Lots of things worked well in CORBA, as long as both sides agree that they're in a CORBA world.

The Web services world is certainly borrowing ideas from CORBA, as CORBA borrowed ideas from earlier technologies. What they're trying to do in Web services is borrow the few ideas in CORBA that actually panned out.

EA I get the distinct impression, Roger, that your attitude is CORBA failed, and Web services has succeeded. Yet CORBA is used for lots of very real things.

CNN, for example, uses CORBA. Most phone systems use CORBA. And the poster-child example of Web services has been Google. It looks to me like CORBA is more of a success than Web services.

RS I totally disagree with that. I would say that relatively few CORBA applications have panned out. Anybody who is investing any money in a CORBA architecture is making a big mistake.

None of the major players that was instrumental in bringing CORBA about is investing in its future. IBM is investing nothing into CORBA. Sun is investing nothing



into CORBA. Microsoft never cared about CORBA. So who is investing in it? Some marginal player someplace.

When you mention Google, you're talking about a very specific, and limited, application. When you look at Web services, you really need to categorize it into one of two types of applications: inter-enterprise or intra-enterprise. Google is an example of inter-enterprise.

My position has always been that inter-enterprise is a marginal area of Web services. It's the one that Microsoft and IBM peddle when they're talking to everybody about this. But the much more important area for Web services—the one that's being used many, many places—is getting different technology systems to interoperate within the same enterprise.

EA Roger made the rather telling statement that Microsoft never looked at CORBA. Could I make a legitimate argument that CORBA failed and Web services “succeeded”—and I'm not admitting that yet—because of the Microsoft hegemony over the world? What I'm suggesting is, had Microsoft supported CORBA, would we not be talking about Web services at all?

RS No, because Microsoft is not what killed CORBA. J2EE killed CORBA. If you want to blame somebody for killing CORBA, blame IBM and Sun, because all the major players that were originally looking at CORBA as their savior technology abandoned it and moved on to J2EE.

TC I actually agree, totally, with Roger on this. But it seems to me that one of the reasons we have a huge wealth of Web services stuff cropping up is because our friends at Microsoft are making it completely trivial to build Web services, in the sense that you simply build .NET implementations and then say, “Hey, I'd like to have the Web interfaces available for these.”

Do you think that's true? Are the tools that are making Web services essentially transparent to the developer responsible for part of why they are so popular and why we're seeing a lot of these services inside the enterprises?

RS There is some truth to that. Certainly, if you look

at the major enterprise players, which are, in my view, BEA, IBM, and Microsoft, they are all doing the best job they can to make it as transparent as possible to use Web services.

They did a similar thing with components. They tried to make it very easy to use them, and the problem was that people really never understood what the fundamental differences were between these technologies: objects, components, and Web services.

In some sense, the transparent ability to make something a Web service is not really a good thing, because making an effective Web service requires a much more in-depth understanding of what it means to be a Web



PHOTOGRAPH BY TOM UPTON

service. It's the same with components. These tools don't give you that. They give you the ability to slap a SOAP interface on top of some code, and that's it.

EA How do you think this is going to affect the evolution of Web services? Given that people are going to use these tools, is this going to result in a huge period of extremely poor architectures because people have just slopped Web services on top of existing architectural solutions?

RS Yes, that's my expectation. We have great tools today for building Web services and virtually no understanding of why, when, and where we should build Web services.

EA I'm curious about your view of the developer's world when building up a system. Clearly you believe that one has to see the boundaries between what you refer to as objects and components and Web services. But do those

differences actually translate in your mind to very specific different implementation technologies? Are objects truly different from components, or is it just a design distinction about the role that something plays in the system?

RS I see it as more a design distinction. Just to give you a simple example: state management. If you have an object, it's perfectly OK to keep state in the object long-term. As long as the object lives, it can have state in it. In a component, you can't do that. You've got to get the state out of there or your component will not scale. None of the tools tells you that. You have to know that, and you have to design the system accordingly.

Just because you can use objects to implement your

ary. But location and environmental boundaries have many implications in terms of security, transactions, and other design issues.

EA There's a very, very strong impression that came out of this article, and that is if I'm going to use components, I would never ever consider using components in something in the same process. But I've talked to a number of people now who have said, "Nonsense, we do that all the time, and it's an important point of our flexibility."

RS Then they're really using the wrong technology for what they're doing. They should just be using object technology for that.

TC No, that's false. One of the elements that defines a

component architecture is the point of interception. This is incredibly useful even if I have things talking within the same process, because it gives me the opportunity to, for example, track invocation patterns without actually having to disturb my architecture at all.

We actually do this for the product that we're building at Silicon Chalk. We transparently introduce a layer of debugging proxies and get all kinds of tracing information that vastly improves our ability to debug the system. We couldn't do that if we were building it out of C++

without having some base class nightmare to deal with.

So the fact that component technology provides a point of interception actually turns out to be an incredibly valuable tool to the developer.

RS There are object systems that provide that as well. You're picking on the shortcomings of a particular language and using that to condemn all object-oriented systems. That's not fair. If you need interception, if that's a useful tool, then you choose an object technology that provides interception.

TC I don't have those choices as a developer out in the real world. Sometimes you have to work in a particular language or system. That's the land that I live in, and that's the reality for most developers as well. Component systems offer me the power that I need to build my



PHOTOGRAPH BY GEORGE BRAINARD

components doesn't mean that objects and components are semantically equivalent. State management is one example, but there are many others. These are design issues, not technology issues.

EA Now you've just introduced the semantic element. There are lots of semantics that objects have—polymorphism, encapsulation, inheritance—which you can sort of build into Web services, perhaps just as I can write object-oriented C, but it's not the same thing.

RS It's not even clear that that's a good idea. In my mind, to have inheritance on top of a Web service is probably a bad idea.

In the "Fuzzy Boundaries" article I said that the defining characteristics that differentiate objects, components, and Web services are location and environmental bound-

product and deliver it to my customers. Now it's true, if I had been programming in Smalltalk, I could go in there and fiddle with the dispatch mechanism. But I don't have that option.

RS That's unfortunate. You chose the wrong language.

TC I chose the only language that made any sense, given the other realities of the world that I deal with. It's nice to talk about distinctions between objects and components as if one could make a completely free choice about how to implement things, but the real world doesn't work that way. As a person who is responsible for actually getting a product out the door and satisfying customers, you can't choose arbitrary technologies because they happen to satisfy purist notions of what is appropriate.

RS If you're saying that you are using one particular aspect of one particular component technology to make up for a regrettable constraint on one particular programming language, then that's OK. Do what you need to do. But just because you are using interception doesn't make it a defining difference between components and objects. That's just a particular artifact of the constraints that you happen to be working under.

EA OK, gentlemen, let's shift gears a little. In the course of this discussion we've hit upon various standardization efforts that have come out or are evolving. For example, a

lot of stuff is happening with WS security and WS transactions, WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration). I'm curious to get Roger's point of view on which of these things are good, and where we should be doing things differently. There are lots of standards out there, and, frankly, they're at least as hard, if not harder, to understand than some of the CORBA specifications were.

RS I agree that the Web services standards are harder to understand than most of the CORBA specifications, but there's one fundamental difference between these specifications and the CORBA ones. The CORBA specifications had to be understood by developers. The Web services standards don't. Nobody needs to understand the Web services standards except for Microsoft and IBM because these standards are about how Microsoft and IBM are going to talk together, not about how the developer is going to do anything.

EA So, nobody is ever going to interact except Microsoft and IBM?

RS The people who are building the platforms are the ones who care about these standards. These standards have no relevance to Joe or Jane Developer, none whatsoever.

TC Do you mean that Joe or Jane Developer is never

The image shows the cover of the October issue of ACM Queue magazine. The top left features the 'acm queue' logo in white on a blue background, with the tagline 'architecting tomorrow's computing' below it. The right side of the cover has a dark, abstract background with a bright, glowing light source. The text 'Coming in the October issue' is written in white. The main title 'Semi-Structured Data' is prominently displayed in large white letters. Below the title, three sub-articles are listed in white text: 'Are ontologies the answer?', 'How XML can help', and 'Lessons learned from the Web'.

going to make use of anything that uses UDDI?

RS They're going to use things that make use of UDDI, but UDDI is going to be at a much lower level than anything they will see.

TC You are saying that they'll never see UDDI?

RS They will see their particular vendors' tools for using UDDI. But they're not going to see UDDI itself. It's like a Web developer worrying about TCP/IP. Yes, they're going to use it, but they're not going to see it.

TC So, again, looking at this from a developer's point of view, I'm going to build a service-oriented architecture and basically you're saying I don't need to know anything about Web services standards, but I need to know about Web services as an abstract idea in order to get my architecture correct.

RS You need to understand architecturally what you need to do to build effective Web services, but as far as how the Web services standards move information around, that's not your problem.

TC What about issues of performance?

RS You'll need to know what your vendor supports. These standards are not about development. They are about interoperability among different vendor platforms.

Look at the standards. Give me one example of a standard that would bubble through to a developer? One API

that has been defined in all the Web services that would bubble back to a developer? There aren't any!

TC It sounds like what you're saying is that the tools that automatically supply Web services interfaces are, in fact, absolutely necessary because they're that insulation between the developers and the underlying protocols. At the same time, they're the downfall that's making it possible to generate poorly architected systems. Two-edged sword?

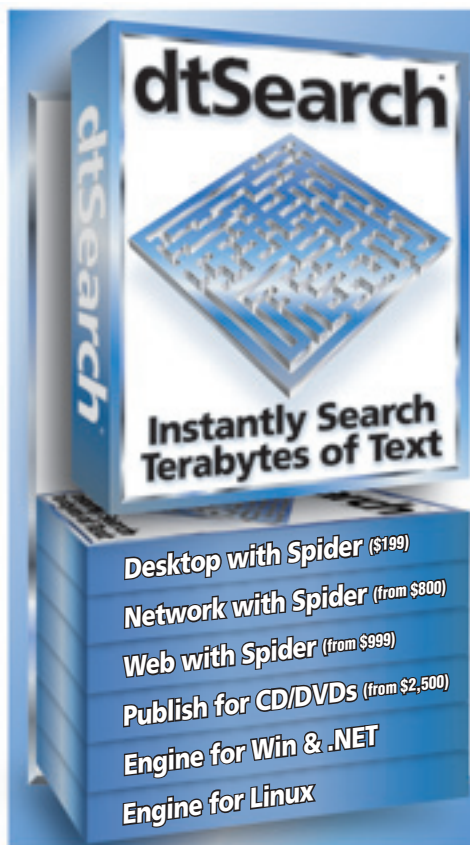
RS Even if you didn't have those tools, it would still be perfectly possible to implement poor systems. In fact, it's probably even easier because you'd have a lot more things you could mess up.

Perhaps you could argue that without these tools, you would have to know so much about Web services that you would be bound to pick up some design smarts somewhere along the path. I think the answer to this dilemma is not to get rid of the tools, but to educate people, through magazines such as *Queue*, that show them the right way to build these things. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

© 2005 ACM 1542-7730/05/0900 \$5.00



NEW Version 7 Terabyte Indexer

The Smart Choice for Text Retrieval® since 1991

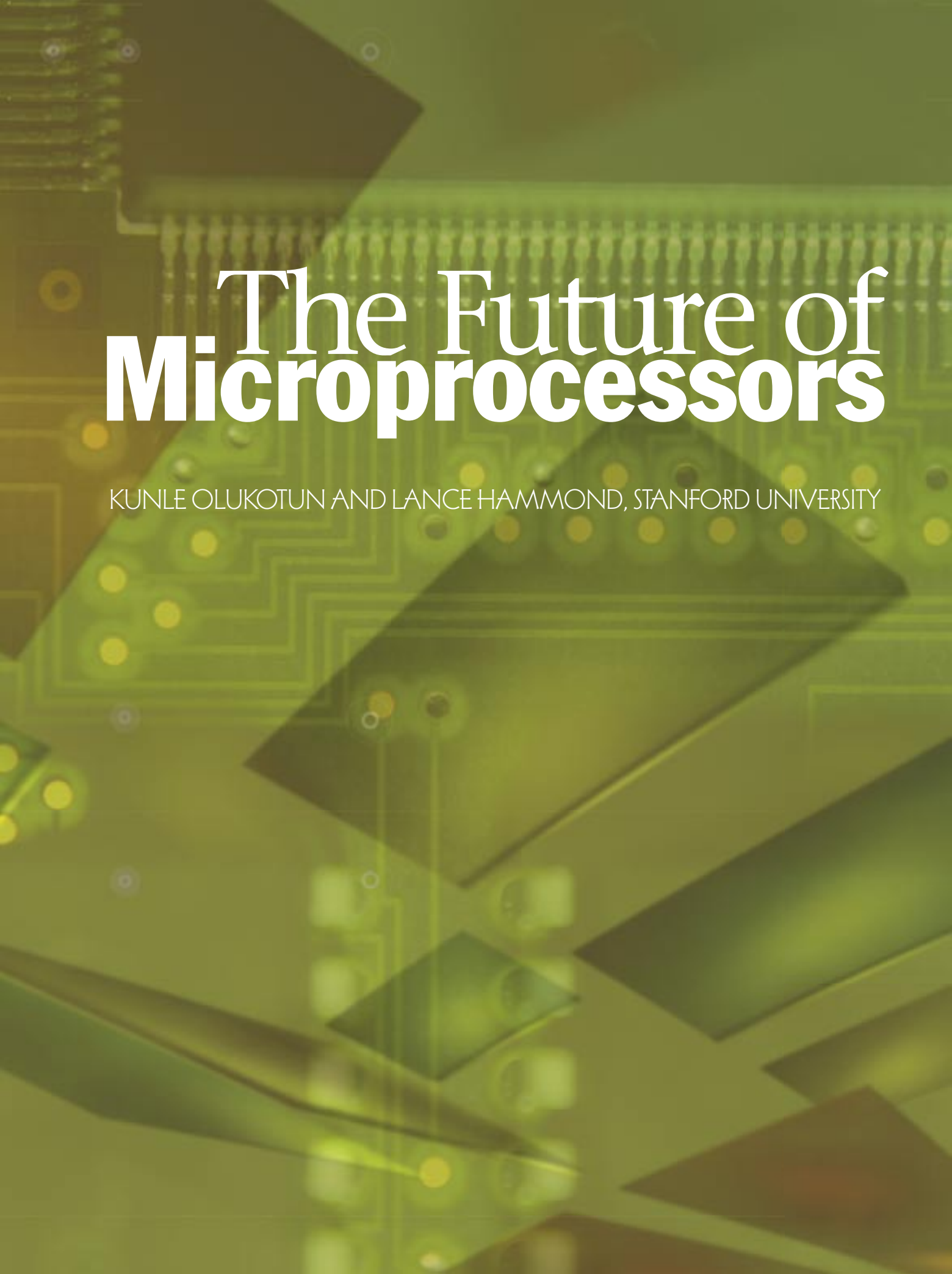
- ◆ over two dozen indexed, unindexed, fielded & full-text search options
- ◆ **highlights hits** in HTML, XML and PDF while displaying embedded **links**, formatting and **images**
- ◆ converts other file types (word processor, database, spreadsheet, email, ZIP, Unicode, etc.) to HTML for display with **highlighted hits**

Reviews of dtSearch

- ◆ "The most powerful document search tool on the market" — *Wired Magazine*
- ◆ "dtSearch ... leads the market" — *Network Computing*
- ◆ "Blindingly fast" — *Computer Forensics: Incident Response Essentials*
- ◆ "A powerful arsenal of search tools" — *The New York Times*
- ◆ "Super fast, super-reliable" — *The Wall Street Journal*
- ◆ "Covers all data sources ... powerful Web-based engines" — *eWEEK*
- ◆ "Searches at blazing speeds" — *Computer Reseller News Test Center*

See www.dtsearch.com for hundreds more reviews & case studies

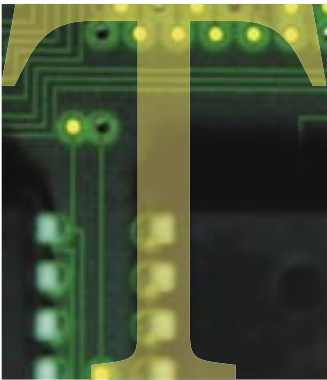
1-800-IT-FINDS • www.dtsearch.com



The Future of **Microprocessors**

KUNLE OLUKOTUN AND LANCE HAMMOND, STANFORD UNIVERSITY

**Chip multiprocessors’
promise of huge
performance gains
is now a reality.**



he performance of microprocessors that power modern computers has continued to increase exponentially over the years for two main reasons. First, the transistors that are the heart of the circuits in all processors and memory chips have simply become faster over time on a course described by Moore’s law,¹ and this directly affects the performance of processors built with those transistors. Moreover, actual processor performance has increased *faster* than Moore’s law would predict,² because processor designers have been able to harness the increasing numbers of transistors available on modern chips to extract more *parallelism* from

software. This is depicted in figure 1 for Intel’s processors.

An interesting aspect of this continual quest for more parallelism is that it has been pursued in a way that has been virtually invisible to software programmers. Since they were invented in the 1970s, microprocessors have continued to implement the conventional von Neumann computational model, with very few exceptions or modifications. To a programmer, each computer consists of a single processor executing a stream of sequential instructions and connected to a monolithic “memory” that holds all of the program’s data. Because the economic benefits of backward compatibility with earlier generations of processors are so strong, hardware designers have essentially been limited to enhancements that have maintained this abstraction for decades. On the memory side, this has resulted in processors with larger cache memories, to keep frequently accessed portions of the conceptual “memory” in small, fast memories that are physically closer to the processor, and large register files to hold more active data values in an



The Future of Microprocessors

extremely small, fast, and compiler-managed region of “memory.”

Within processors, this has resulted in a variety of modifications designed to achieve one of two goals: increasing the number of instructions from the processor’s instruction sequence that can be issued on every cycle, or increasing the clock frequency of the processor faster than Moore’s law would normally allow. Pipelining of individual instruction execution into a sequence of stages has allowed designers to increase clock rates as instructions have been sliced into larger numbers of increasingly small steps, which are designed to reduce the amount of logic that needs to switch during every clock cycle. Instructions that once took a few cycles to execute in the 1980s now often take 20 or more in today’s leading-edge processors, allowing a nearly proportional increase in the possible clock rate.

Meanwhile, superscalar processors were developed to execute multiple instructions from a single, conventional instruction stream on each cycle. These function by dynamically examining sets of instructions from the instruction stream to find ones capable of parallel execution on each cycle, and then executing them, often out of order with respect to the original program.

Both techniques have flourished because they allow instructions to execute more quickly while maintaining the key illusion for programmers that all instructions are actually being executed sequentially and in order, instead of overlapped and out of

order. Of course, this illusion is not absolute. Performance can often be improved if programmers or compilers adjust their instruction scheduling and data layout to map more efficiently to the underlying pipelined or parallel architecture and cache memories, but the important point is that old or untuned code will still execute correctly on the architecture, albeit at less-than-peak speeds.

Unfortunately, it is becoming increasingly difficult for processor designers to continue using these techniques to enhance the speed of modern processors. Typical instruction streams have only a limited amount of usable parallelism among instructions,³ so superscalar processors that can issue more than about four instructions per cycle achieve very little additional benefit on most applications. Figure 2 shows how effective real Intel processors have been at extracting instruction parallelism over time. There is a flat region before instruction-level parallelism was pursued intensely, then a steep rise as parallelism was utilized usefully, followed by a tapering off in recent years as the available parallelism has become fully exploited.

Complicating matters further, building superscalar processor cores that can exploit more than a few instructions per cycle becomes very expensive, because the complexity of all the additional logic required to find parallel instructions dynamically is approximately proportional to the square of the number of instructions that can be issued simultaneously. Similarly, pipelining past about 10-20 stages is difficult because each pipeline stage becomes too short to perform even a minimal amount of

Intel Performance Over Time

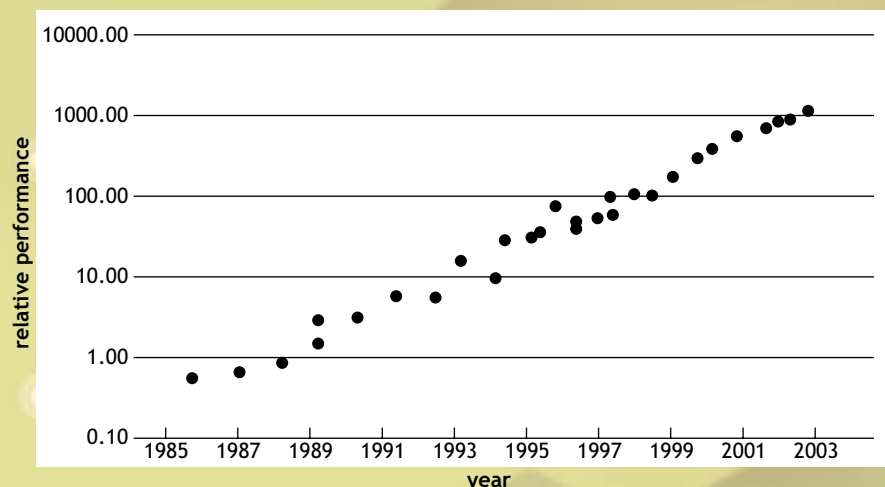


FIG 1

logic, such as adding two integers together, beyond which the design of the pipeline is significantly more complex. In addition, the circuitry overhead from adding pipeline registers and bypass path multiplexers to the existing logic combines with performance losses from events that cause pipeline state to be flushed, primarily branches. This overwhelms any potential performance gain from deeper pipelining after about 30 stages.

Further advances in both superscalar issue and pipelining are also limited by the fact that they require ever-larger numbers of transistors to be integrated into the high-speed central logic within each processor core—so many, in fact, that few companies can afford to hire enough engineers to design and verify these processor cores in reasonable amounts of time. These trends have slowed the advance in processor performance somewhat and have forced many smaller vendors to forsake the high-end processor business, as they could no longer afford to compete effectively.

Today, however, all progress in conventional processor core development has essentially stopped because of a simple physical limit: power. As processors were pipelined and made increasingly superscalar over the course of the past two decades, typical high-end microprocessor power went from less than a watt to over 100 watts. Even though each silicon process generation promised a reduction in power, as the ever-smaller transistors required less power to switch, this was true in practice only when existing designs were simply “shrunk” to use the new

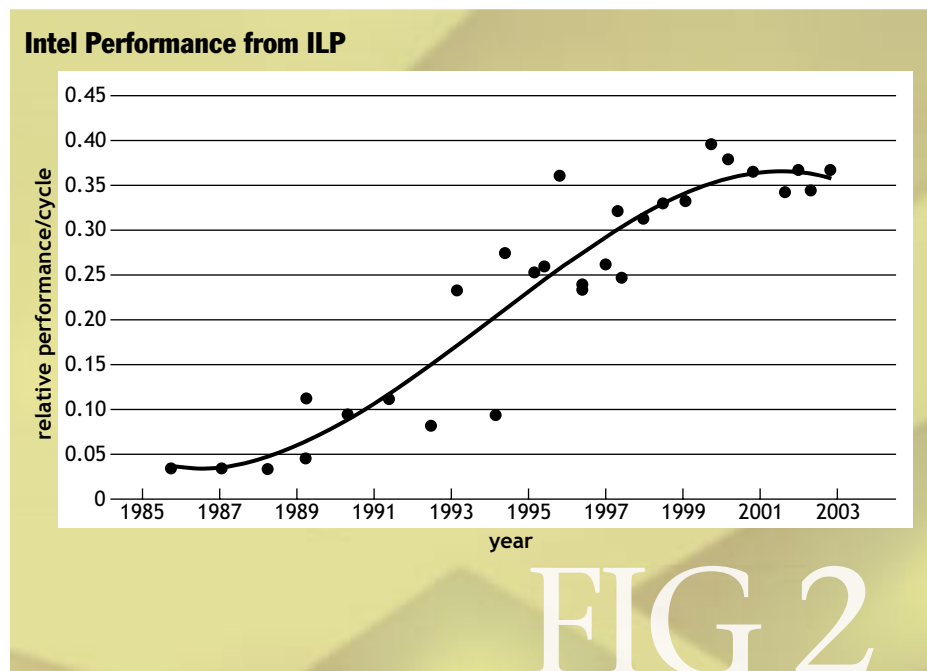
process technology. Processor designers, however, kept using more transistors in their cores to add pipelining and superscalar issue, and switching them at higher and higher frequencies. The overall effect was that exponentially more power was required by each subsequent processor generation (as illustrated in figure 3).

Unfortunately, cooling technology does not scale exponentially nearly as easily. As a result, processors went from needing no heat sinks in the 1980s, to moderate-size heat sinks in the 1990s, to today’s monstrous heat sinks, often with one or more dedicated fans to increase airflow over the processor. If these trends were to continue, the next generation of microprocessors would require very exotic cooling solutions, such as dedicated water cooling, that are economically impractical in all but the most expensive systems.

The combination of limited instruction parallelism suitable for superscalar issue, practical limits to pipelining, and a “power ceiling” limited by practical cooling limitations has limited future speed increases within conventional processor cores to the basic Moore’s law improvement rate of the underlying transistors. This limitation is already causing major processor manufacturers such as Intel and AMD to adjust their marketing focus away from simple core clock rate.

Although larger cache memories will continue to improve performance somewhat, by speeding access to the single “memory” in the conventional model, the simple fact is that without more radical changes in processor design, microprocessor performance increases

will slow dramatically in the future. Processor designers must find new ways to *effectively* utilize the increasing transistor budgets in high-end silicon chips to improve performance in ways that minimize both additional power usage *and* design complexity. The market for microprocessors has become stratified into areas with different performance requirements, so it is useful to examine the problem from the point of view of these different performance requirements.



The Future of Microprocessors

THROUGHPUT PERFORMANCE IMPROVEMENT

With the rise of the Internet, the need for servers capable of handling a multitude of independent requests arriving rapidly over the network has increased dramatically. Since individual network requests are typically completely independent tasks, whether those requests are for Web pages, database access, or file service, they are typically spread across many separate computers built using high-performance conventional microprocessors (figure 4a), a technique that has been used at places like Google for years to match the overall computation throughput to the input request rate.⁴

As the number of requests increased over time, more servers were added to the collection. It has also been possible to replace some or all of the separate servers with multiprocessors. Most existing multiprocessors consist of two or more separate processors connected using a common bus, switch hub, or network to shared memory and I/O devices. The overall system can usually be physically smaller and use less power than an equivalent set of uniprocessor systems because physically large components such as memory, hard drives, and power supplies can be shared by some or all of the processors.

Pressure has increased over time to achieve more performance per unit volume of data-center space and per watt, since data centers have finite room for servers and their electric bills can be staggering. In response, the server manufacturers have tried to save space by adopting denser server packaging

solutions, such as blade servers and switching to multiprocessors that can share components. Some power reduction has also occurred through the sharing of more power-hungry components in these systems. These short-term solutions are reaching their practical limits, however, as systems are reaching the maximum component density that can still be effectively air-cooled. As a result, the next stage of development for these systems involves a new step: the CMP (chip multiprocessor).⁵

The first CMPs targeted toward the server market implement two or more conventional superscalar processors together on a single die.^{6,7,8,9} The primary motivation for this is reduced volume—multiple processors can now fit in the space where formerly only one could, so overall performance per unit volume can be increased. Some savings in power also occurs because all of the processors on a single die can share a single connection to the rest of the system, reducing the amount of high-speed communication infrastructure required, in addition to the sharing possible with a conventional multiprocessor. Some CMPs, such as the first ones announced from AMD and Intel, share only the system interface between processor cores (illustrated in figure 4b), but others share one or more levels of on-chip cache (figure 4c), which allows interprocessor communication between the CMP cores without off-chip accesses.

Further savings in power can be achieved by taking advantage of the fact that while server workloads require high throughput, the *latency* of each request is generally

Intel Power Over Time

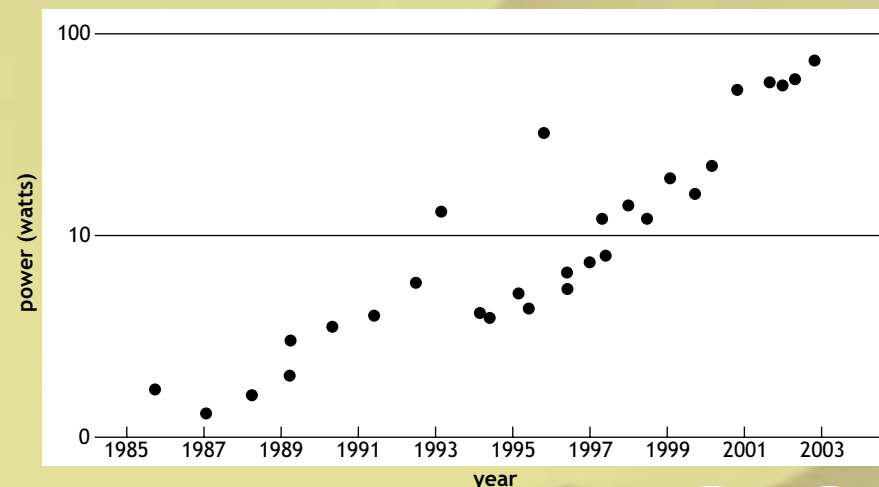


FIG 3

not as critical.¹⁰ Most users will not be bothered if their Web pages take a fraction of a second longer to load, but they will complain if the Web site drops page requests because it does not have enough throughput capacity. A CMP-based system can be designed to take advantage of this situation.

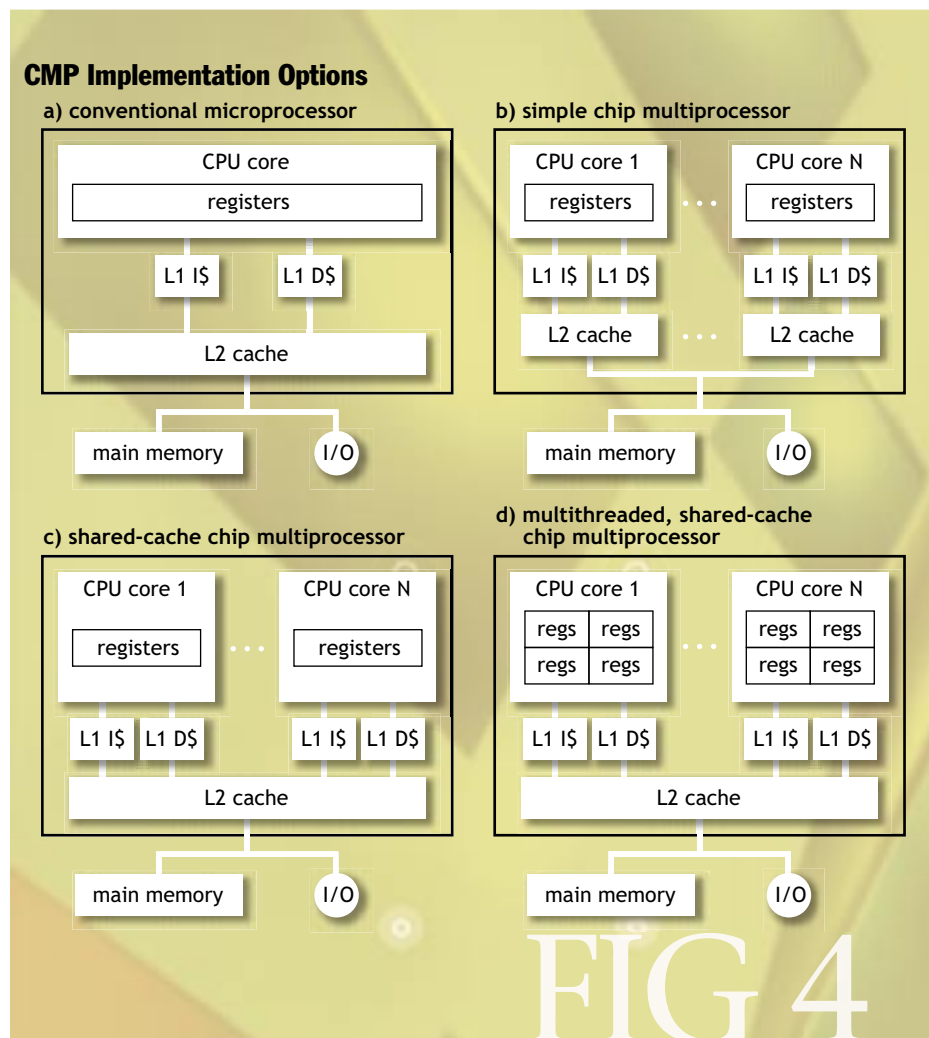
When a two-way CMP replaces a uniprocessor, it is possible to achieve essentially the same or better throughput on server-oriented workloads with just *half* of the original clock speed. Each request may take up to twice as long to process because of the reduced clock rate. With many of these applications, however, the slowdown will be much less, because request processing time is more often limited by memory or disk performance than by processor performance. Since two requests can now be processed simultaneously, however, the overall throughput will now be the same or better, unless there is serious contention for the same memory or disk resources.

Overall, even though performance is the same or only a little better, this adjustment is still advantageous at the system level. The lower clock rate allows us to design the system with a significantly lower power supply voltage, often a nearly linear reduction. Since power is proportional to the *square* of the voltage, however, the power required to obtain the original performance is much lower—usually about half (half of the voltage squared = a quarter of the power, per processor, so the power required for both processors together is about half), although the potential savings could be limited by static power dissipation and any minimum voltage levels required by the underlying transistors.

For throughput-oriented workloads, even more power/performance and performance/chip area can be achieved by taking the “latency is unimportant” idea to its extreme and building the CMP with many small cores instead of a few large ones. Because typical server workloads have very

low amounts of instruction-level parallelism and many memory stalls, most of the hardware associated with superscalar instruction issue is essentially wasted for these applications. A typical server will have tens or hundreds of requests in flight at once, however, so there is enough work available to keep many processors busy simultaneously.

Therefore, replacing each large, superscalar processor in a CMP with several small ones, as has been demonstrated successfully with the Sun Niagara,¹¹ is a winning policy. Each small processor will process its request more slowly than a larger, superscalar processor, but this latency slowdown is more than compensated for by the fact that the same chip area can be occupied by a much larger number of processors—about four times as many, in the case



The Future of Microprocessors

of Niagara, which has eight single-issue SPARC processor cores in a technology that can hold only a pair of superscalar UltraSPARC cores.

Taking this idea one step further, still more latency can be traded for higher throughput with the inclusion of multithreading logic within each of the cores.^{12,13,14} Because each core tends to spend a fair amount of time waiting for memory requests to be satisfied, it makes sense to assign each core several threads by including multiple register files, one per thread, within each core (figure 4d). While some of the threads are waiting for memory to respond, the processor may still execute instructions from the others.

Larger numbers of threads can also allow each processor to send more requests off to memory in parallel, increasing the utilization of the highly pipelined memory systems on today's processors. Overall, threads will typically have a slightly longer latency, because there are times when all are active and competing for the use of the processor core. The gain from performing computation during memory stalls and the ability to launch numerous memory accesses simultaneously more than compensates for this longer latency on systems such as Niagara, which has four threads per processor or 32 for the entire chip, and Pentium chips with Intel's Hyperthreading, which allows two threads to share a Pentium 4 core.

LATENCY PERFORMANCE IMPROVEMENT

The performance of many important applications is measured in terms of the execution latency of individual tasks instead of high overall throughput of many essentially unrelated tasks. Most desktop processor applications still fall in this category, as users are generally more concerned with their computers responding to their commands as quickly as possible than they are with their computers' ability to handle many commands simultaneously, although this situation is changing slowly over time as more applications are written to include many "background" tasks. Users of many other computation-bound applications, such as most simulations and compilations,

are typically also more interested in how long the programs take to execute than in executing many in parallel.

Multiprocessors can speed up these types of applications, but it requires effort on the part of programmers to break up each long-latency thread of execution into a large number of smaller threads that can be executed on many processors in parallel, since automatic parallelization technology has typically functioned only on Fortran programs describing dense-matrix numerical computations. Historically, communication between processors was generally slow in relation to the speed of individual processors, so it was critical for programmers to ensure that threads running on separate processors required only minimal communication with each other.

Because communication reduction is often difficult, only a small minority of users bothered to invest the time and effort required to parallelize their programs in a way that could achieve speedup, so these techniques were taught only in advanced, graduate-level computer science courses. Instead, in most cases programmers found that it was just easier to wait for the next generation of uniprocessors to appear and speed up their applications for "free" instead of investing the effort required to parallelize their programs. As a result, multiprocessors had a hard time competing against uniprocessors except in very large systems, where the target performance simply exceeded the power of the fastest uniprocessors available.

With the exhaustion of essentially all performance gains that can be achieved for "free" with technologies such as superscalar dispatch and pipelining, we are now entering an era where programmers *must* switch to more parallel programming models in order to exploit multiprocessors effectively, if they desire improved single-program performance. This is because there are only three real "dimensions" to processor performance increases beyond Moore's law: clock frequency, superscalar instruction issue, and multiprocessing. We have pushed the first two to their logical limits and must now embrace multiprocessing, even if it means that programmers will be forced to change to a parallel programming model to achieve the highest possible performance.

Conveniently, the transition from multiple-chip systems to chip multiprocessors greatly simplifies the problems traditionally associated with parallel programming. Previously it was necessary to minimize communication between independent threads to an extremely low level, because each communication could require hundreds or even thousands of processor cycles. Within any CMP with a shared on-chip cache memory, however, each communication event typically takes just a handful

of processor cycles. With latencies like these, communication delays have a much smaller impact on overall system performance. Programmers must still divide their work into parallel threads, but do not need to worry nearly as much about ensuring that these threads are highly independent, since communication is relatively cheap. This is not a complete panacea, however, because programmers must still structure their inter-thread synchronization correctly, or the program may generate incorrect results or deadlock, but at least the performance impact of communication delays is minimized.

Parallel threads can also be much smaller and still be effective—threads that are only hundreds or a few thousand cycles long can often be used to extract parallelism with these systems, instead of the millions of cycles long threads typically necessary with conventional parallel machines. Researchers have shown that parallelization of applications can be made even easier with several schemes involving the addition of *transactional* hardware to a CMP.^{15,16,17,18,19} These systems add buffering logic that lets threads attempt to execute in parallel, and then dynamically determines whether they are actually parallel at runtime. If no inter-thread dependencies are detected at runtime, then the threads complete normally. If dependencies exist, then the buffers of some threads are cleared and those threads are restarted, dynamically serializing the threads in the process.

Such hardware, which is only practical on tightly coupled parallel machines such as CMPs, eliminates the need for programmers to determine whether threads are parallel as they parallelize their programs—they need only choose *potentially* parallel threads. Overall, the shift from conventional processors to CMPs should be less traumatic for programmers than the shift from conventional processors to multichip multiprocessors, because of the short CMP communication latencies and enhancements such as transactional memory, which should be commercially available within the next few years. As a result, this paradigm shift should be within the range of what is feasible for “typical” programmers, instead of being limited to graduate-level computer science topics.

HARDWARE ADVANTAGES

In addition to the software advantages now and in the future, CMPs have major advantages over conventional uniprocessors for hardware designers. CMPs require only a fairly modest engineering effort for each generation of processors. Each member of a family of processors just requires the stamping down of additional copies of the core processor and then making some modifications to

relatively slow logic connecting the processors together to accommodate the additional processors in each generation—and *not* a complete redesign of the high-speed processor core logic. Moreover, the system board design typically needs only minor tweaks from generation to generation, since externally a CMP looks essentially the same from generation to generation, even as the number of processors within it increases.

The only real difference is that the board will need to deal with higher I/O bandwidth requirements as the CMPs scale. Over several silicon process generations, the savings in engineering costs can be significant, because it is relatively easy to stamp down a few more cores each time. Also, the same engineering effort can be amortized across a large family of related processors. Simply varying the numbers *and* clock frequencies of processors can allow essentially the same hardware to function at many different price/performance points.

AN INEVITABLE TRANSITION

As a result of these trends, we are at a point where chip multiprocessors are making significant inroads into the marketplace. Throughput computing is the first and most pressing area where CMPs are having an impact. This is because they can improve power/performance results right out of the box, without any software changes, thanks to the large numbers of independent threads that are available in these already multithreaded applications. In the near future, CMPs should also have an impact in the more common area of latency-critical computations. Although it is necessary to parallelize most latency-critical software into multiple parallel threads of execution to really take advantage of a chip multiprocessor, CMPs make this process easier than with conventional multiprocessors, because of their short interprocessor communication latencies.

Viewed another way, the transition to CMPs is inevitable because past efforts to speed up processor architectures with techniques that do not modify the basic von Neumann computing model, such as pipelining and superscalar issue, are encountering hard limits. As a result, the microprocessor industry is leading the way to multicore architectures; however, the full benefit of these architectures will not be harnessed until the software industry fully embraces parallel programming. The art of multiprocessor programming, currently mastered by only a small minority of programmers, is more complex than programming uniprocessor machines and requires an understanding of new computational principles, algorithms, and programming tools. ☐

The Future of Multiprocessors

REFERENCES

1. Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics* (April): 114–117.
2. Hennessy, J. L., and Patterson, D. A. 2003. *Computer Architecture: A Quantitative Approach*, 3rd Edition, San Francisco, CA: Morgan Kaufmann Publishers.
3. Wall, D. W. 1993. *Limits of Instruction-Level Parallelism*, WRL Research Report 93/6, Digital Western Research Laboratory, Palo Alto, CA.
4. Barroso, L., Dean, J., and Hoezle, U. 2003. Web search for a planet: the architecture of the Google cluster. *IEEE Micro* 23 (2): 22–28.
5. Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. 1996. The case for a single chip multiprocessor. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*: 2–11.
6. Kapil, S. 2003. UltraSPARC Gemini: Dual CPU Processor. In *Hot Chips 15* (August), Stanford, CA; <http://www.hotchips.org/archives/>.
7. Maruyama, T. 2003. SPARC64 VI: Fujitsu's next generation processor. In *Microprocessor Forum* (October), San Jose, CA.
8. McNairy, C., and Bhatia, R. 2004. Montecito: the next product in the Itanium processor family. In *Hot Chips 16* (August), Stanford, CA; <http://www.hotchips.org/archives/>.
9. Moore, C. 2000. POWER4 system microarchitecture. In *Microprocessor Forum* (October), San Jose, CA.
10. Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzky, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture* (June): 282–293.
11. Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro* 25 (2): 21–29.
12. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. 1990. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing* (June): 1–6.
13. Laudon, J., Gupta, A., and Horowitz, M. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*: 308–316.
14. Tullsen, D. M., Eggers, S. J., and Levy, H. M. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June): 392–403.
15. Hammond, L., Carlstrom, B. D., Wong, V., Chen, M., Kozyrakis, C., and Olukotun, K. 2004. Transactional coherence and consistency: simplifying parallel hardware and software. *IEEE Micro* 24 (6): 92–103.
16. Hammond, L., Hubbert, B., Siu, M., Prabhu, M., Chen, M., and Olukotun, K. 2000. The Stanford Hydra CMP. *IEEE Micro* 20 (2): 71–84.
17. Krishnan, V., and Torrellas, J. 1999. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers* 48 (9): 866–880.
18. Sohi, G., Breach, S., and Vijaykumar, T. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture* (June): 414–425.
19. Steffan, J. G., and Mowry, T. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture* (February): 2–13.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

KUNLE OLUKOTUN is an associate professor of electrical engineering and computer science at Stanford University, where he led the Stanford Hydra single-chip multiprocessor research project, which pioneered multiple processors on a single silicon chip. He founded Afara Websystems to develop commercial server systems with chip multiprocessor technology. Afara was acquired by Sun Microsystems, and the Afara microprocessor technology is now called Niagara. Olukotun is involved in research in computer architecture, parallel programming environments, and scalable parallel systems.

LANCE HAMMOND is a postdoctoral fellow at Stanford University. As a Ph.D. student, Hammond was the lead architect and implementer of the Hydra chip multiprocessor. The goal of Hammond's recent work on transactional coherence and consistency is to make parallel programming accessible to the average programmer.

© 2005 ACM 1542-7730/05/0900 \$5.00

DEVELOP YOUR **EXPERTISE**



"My advice to you is this:
If you don't keep up,
you're gone."

Gerald Weinberg, Industry Pioneer, SD West 2005 keynote address

SD **BEST** 2005 **PRACTICES** CONFERENCE & EXPO

September 26 – 29, Boston
Hynes Convention Center

Over 150 classes and tutorials in:

- C++ Technical Track -new at SD Best Practices
- Build & Deploy
- Design & Architecture
- People, Project & Teams
- Process & Methods
- Requirements & Analysis
- Testing & Quality

Visit www.sdexpo.com
for more information



CMP

United Business Media



FOCUS

Multiprocessors

Chip multiprocessors
have introduced a
new dimension in
scaling for application
developers, operating
system designers, and
deployment specialists.

EXTREME



Software Scaling

The advent of SMP (symmetric multiprocessing) added a new degree of scalability to computer systems. Rather than deriving additional performance from an incrementally faster microprocessor, an SMP system leverages multiple processors to obtain large gains in total system performance. Parallelism in software allows multiple jobs to execute concurrently on the system, increasing system throughput accordingly. Given sufficient software parallelism, these systems have proved to scale to several hundred processors.

More recently, a similar phenomenon is occurring at the chip level. Rather than pursue diminishing returns by increasing individual processor performance, manufacturers are producing chips with multiple processor cores on a single die. (See “The Future of Microprocessors,” by Kunle Olukotun and Lance Hammond, in this issue.) For example, the AMD Opteron¹ processor now uses two entire processor cores per die, providing almost double the performance of a single core chip. The Sun Niagara² processor, shown in figure

RICHARD MCDOUGALL, SUN MICROSYSTEMS

EXTREME Software Scaling

1, uses eight cores per die, where each core is further multiplexed with four hardware threads each.

These new CMPs (chip multiprocessors) are bringing what was once a large multiprocessor system down to the chip level. A low-end four-chip dual-core Opteron machine presents itself to software as an eight-processor system, and in the case of the Sun Niagara processor with eight cores and four threads per core, a single chip presents itself to software as a 32-processor system. As a result, the ability of system and application software to exploit multiple processors or threads simultaneously is

becoming more important than ever. As CMP hardware progresses, software is required to scale accordingly to fully exploit the parallelism of the chip.

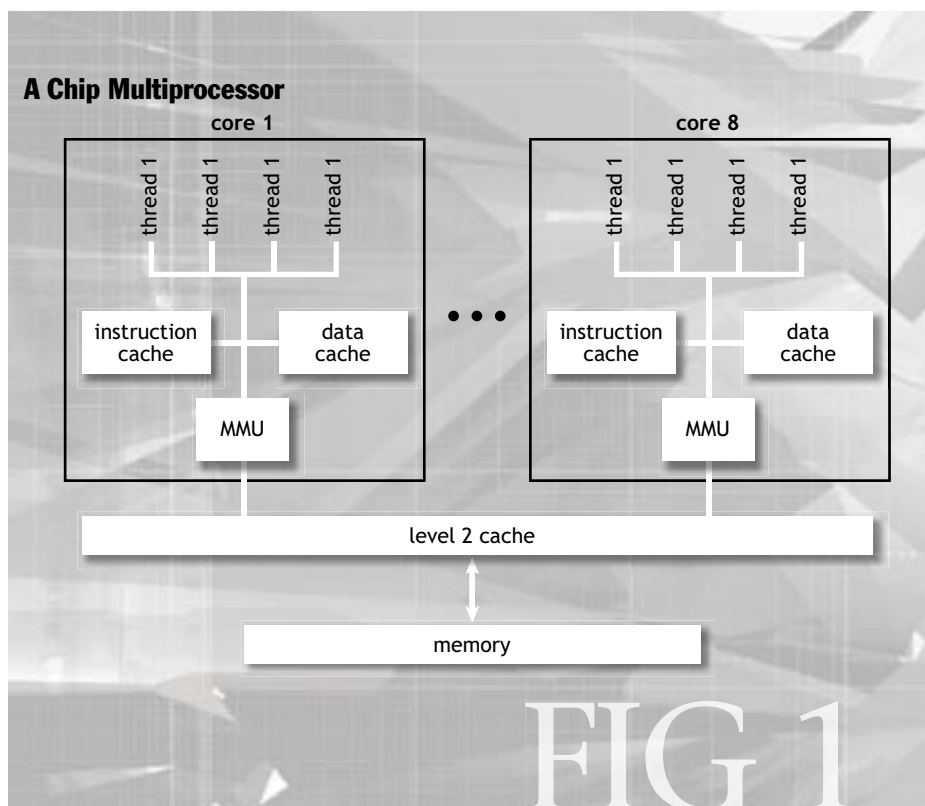
Thus, bringing this degree of parallelism down to the chip level represents a significant change to the way we think about scaling. Since the cost of a CMP system is close to that of recent low-end uniprocessor systems, it's inevitable that even the cheapest desktops and servers will be highly threaded. Techniques used to scale application and system software on large enterprise-level SMP systems will now frequently be leveraged to provide scalability even for single-chip systems. We need to consider the effects of the change in the degree of scaling on the way we architect applications, on which operating system we choose, and on the techniques we use to deploy applications—even at the low end.

CMP: JUST A COST-EFFECTIVE SMP?

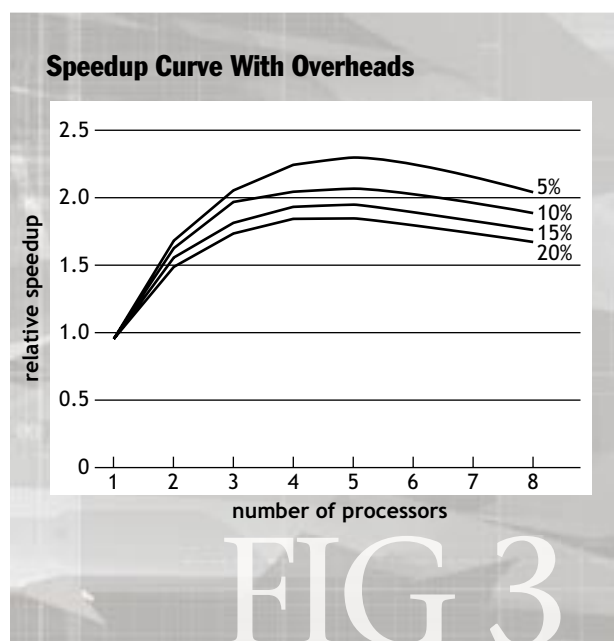
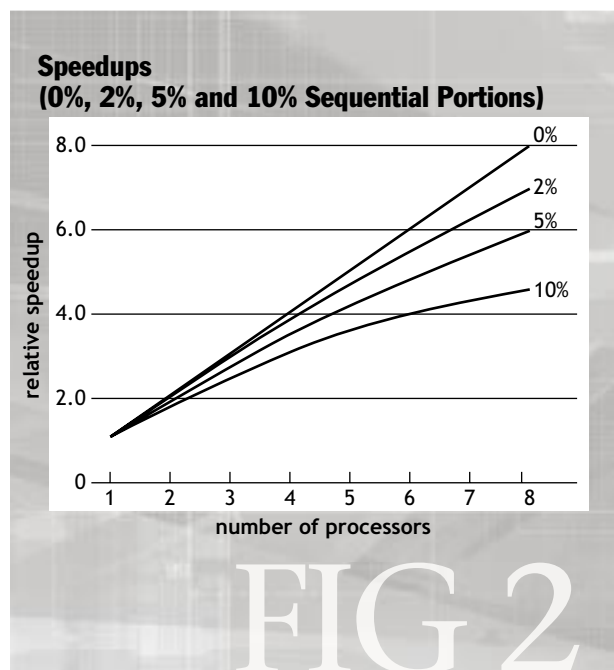
A simplistic view of a CMP system is that it appears to software as an SMP system with the number of processors equal to the number of threads in the chip, each with slightly reduced processing capability. Since each hardware thread is sharing the resources of a single processor core, each thread has some fraction of the core's overall performance. Thus, an eight-core chip with 32 hardware

threads running at 1 GHz may be somewhat crudely approximated as an SMP system with thirty-two 250-MHz processors. The effect on software is often a subtle trade-off in per-thread latency for a significant increase of throughput. For a throughput-oriented workload with many concurrent requests (such as a Web server), the marginal increase in response time is virtually negligible, but the increase in system throughput is an order of magnitude over a non-CMP processor of the same clock speed.

There are, however, more subtle differences between a CMP system and an SMP system. If threads or cores within a CMP pro-



cessor share important resources, then some threads may impact the performance of other threads. For example, when multiple threads share a single core and therefore share first-level memory caches, the performance of a given thread may vary depending on what the other threads, of the same core, are doing with the first thread's data in the cache. Yet, in another similar case, a thread



may actually gain if the other threads are constructively sharing the cache, since useful data may be brought into the cache by threads other than the first. This is covered in more detail later as we explore some of the potential operating system optimizations.

SCALING THE SOFTWARE

The performance of system software ideally scales proportionally with the number of processors in the system. There are, however, factors that limit the speedup.

Amdahl's law³ defines scalability as the speedup of a parallel algorithm, effectively limited by the number of operations that must be performed sequentially (i.e., its *serial fraction*), as shown in figure 2. If 10 percent of a parallel program involves serial code, the maximum speedup that can be attained is three, using four processors (75 percent of linear), reducing to only 4.75 when the processor count increases to eight (only 59 percent of linear). Amdahl's law tells us that the serial fraction places a severe constraint on the speedup as the number of processors increase.

In addition, software typically incurs overhead as a result of communication and distribution of work to multiple processors. This results in a scaling curve where the performance peaks and then begins to degrade (see figure 3).

Since most operating systems and applications contain a certain amount of sequential code, a possible conclusion of Amdahl's law is that it is not cost effective to build systems with large numbers of processors because sufficient speedup will never be produced. Over the past decade, however, the focus has been on reducing the serial fraction within hardware architectures, operating systems, middleware, and application software. Today, it is possible to scale system software and applications on the order of 100 processors on an SMP system. Figure 4 shows the results for a series of scaling benchmarks that were performed using database workloads on a large SMP configuration. These application benchmarks were performed on a single-system image by measuring throughput as the number of processors was increased.

INTRA- OR INTER-MACHINE SCALE?

Software scalability for these large SMP machines has historically been obtained through rigorous focus on intra-machine *scalability* within one large instance of the application within a single operating system. A good example is a one-tier enterprise application such as SAP. The original version of SAP used a single and large monolithic application server. The application instance

EXTREME Software Scaling

obtains its parallelism from the many concurrent requests from users. Providing there are no major serialization points between the users, the application will naturally scale. The focus on scaling these applications has been to remove these serialization points within the applications.

More recently, because of the economics of low-end systems, the focus has been on leveraging inter-machine scaling, using low-cost commodity one- to two-processor servers. Some applications can be made to scale without requiring large, expensive SMP systems by running multiple instances in parallel on separate one- to two-processor systems, resulting in good overall throughput. Applications can be designed to scale this way by moving all shared state to a shared back-end service, like a database. Many one- to two-processor systems are configured as mid-tier application servers, communicating to an intra-machine scaled database system. The shift in focus to one- to two-processor hardware has removed much of the pressure to design intra-machine scalability into the software.

The compelling features of CMP—low power, extreme density, and high throughput—match this space well, mandating a revised focus on intra-machine scalability.

IMPACT OF CMP ON APPLICATION DEVELOPERS

The most significant impact for application developers is the requirement to scale. The minimum scaling requirement has been raised from 1-4 processors to 32 today, and will likely increase again in the near future.

BUILDING SCALABLE APPLICATIONS

Engineering scalable code is challenging, but the performance wins are huge. The data in the scaling curves for Oracle and DB2 in figure 4 show the rewards, from a great deal of performance tuning to optimization for scaling. According to Amdahl's law, scaling software requires minimization of the serial fraction of the workload. In many commercial systems, natural parallelism comes from the many concurrent users of the system.

The simple first-order scaling bottlenecks (those with a large serial fraction) typically come from contention for shared resources, such as:

- **Networks or interconnects.** Bandwidth limitations on interconnects between portions of the system—for example, an ingress network on the Web servers, tier-1 and -2 networks for SQL traffic, or a SAN (storage area network).
- **CPU/Memory.** Queuing for CPU or waiting for page faults as a result of resource starvation.

Scaling of Throughput-Oriented Workloads on SMP Hardware

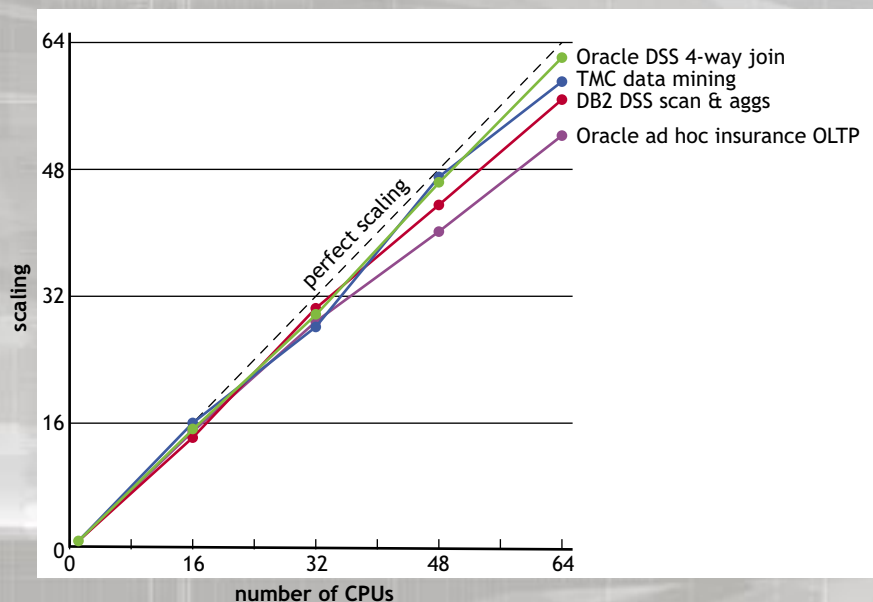


FIG 4

- **I/O throughput.** Insufficient capacity for disk I/O operations or bandwidth.

The more interesting problems result from intrinsic application design. These problems manifest from serial operations within the application or the operating environment. They are often much harder to identify without good observation tools, because rather than showing up as an easy-to-detect overloaded resource (such as out of CPU), they often exhibit growing amounts of idle resource as load is increased.

Here's a common example. We were recently asked to help with a scaling problem on a large online e-commerce system. The application consisted of thousands of users performing payment transactions from a Web application. As load increased, the latency became unacceptable. The application was running on a large SMP system and database, both of which were known to scale well. There was no clear indicator of where in the system the problem occurred. As load was increased, the system CPU resources became *more* idle. It turned out that there was a single table at the center of all the updates, and the locking strategy for the table became the significant serial fraction of the workload. User transactions were simply waiting for updates to the table. The solution was to break up the table so that concurrent inserts could occur, thus reducing the serial fraction and increasing scalability.

For CMP, we need to pay attention to what might limit scaling within one application instance, since we now need to scale in the order of tens of threads, increasing to the order of 100 in the near future.

WRITING SCALABLE LOW-LEVEL CODE

Many middleware applications (such as databases, application servers, or transaction systems) require special attention to scale. Here are a few of the common techniques that may serve as a general guideline.

Scalable algorithms. Many algorithms become less efficient as the size of the problem set increases. For example, an algorithm that searches for an object using a linear list will increase the amount of CPU required as the size of the list increases, potentially at a super-linear rate. Selecting good algorithms that optimize for the common case is of key importance.

Locking. Locking strategies have significant impact on scalability. As concurrency increases, the number of threads attempting to lock an object or region increases, resulting in compounding contention as the lock becomes "hotter." In modern systems, an optimal approach is to provide fine-grained locking using a lock per object where possible. There are also several

approaches to making the reader side of code lock-free at the expense of some memory waste or increased writer-side cost.

Cache line sharing. Multiprocessor and CMP systems use hardware coherency algorithms to keep data consistent between different pipelines. This can have a significant effect on scaling. For example, a latency penalty may result if one processor updates a memory object within its cache, which is also accessed from another processor. The cache location will be invalidated because of the cache coherency hardware protocol, which ensures only one version of the data exists. In a CMP system, multiple threads typically access a single first-level cache; thus, colocating data that will be accessed within a single core may be appropriate.

Pools of worker threads. A good approach for concurrency is to use a pool of worker threads; a general-purpose, multithreaded engine can be used to process an aggregate set of work events. Using this model, an application gives discrete units of work to the engine and lets the engine process them in parallel. The worker pool provides a flexible mechanism to balance the work events across multiple processors or hardware threads. The operating system can automatically tune the concurrency of the application to meet the topology of the underlying hardware architecture.

Memory allocators. Memory allocators pose a significant problem to scaling. Almost every code needs to allocate and free data structures, and typically does so via a central system-provided memory allocator. Unfortunately, very few memory allocators scale well. The few that do include the open source Hoard, Solaris 10's libumem slab allocator, and MicroQuill's SmartHeap. It's worth paying attention to more than one dimension of scalability: different allocators have different properties in light of the nature of allocation/deallocation requests.

CONDUCT SCALABILITY EXPERIMENTS EARLY AND OFTEN

Time has shown that the most efficient way of driving out scaling issues from an application is to perform scaling studies. Given the infinite space in which optimizations can be made, it is important to follow a methodology to prioritize the most important issues.

Modeling techniques can be used to mathematically predict response times and potential scaling bottlenecks in complex systems. They are often used for predicting the performance of hardware, to assist with design trade-off analysis. Modeling software, however, requires intimate knowledge of the software algorithms, code paths, and system service latencies. The time taken to construct

EXTREME Software Scaling

a model and validate all assumptions is often at odds with running scaling tests.

A well-designed set of scaling experiments is key to understanding the performance characteristics of an application, and with proper observation instrumentation, it is easy to pinpoint key issues. Scalability prediction and analysis should be done as early as possible in the development cycle. It's often much harder to retrofit scalability improvements to an existing architecture. Consider scalability as part of the application architecture and design.

Key items to include in scalability experiments are:

- **Throughput versus number of threads/processors.** Does the throughput scale close to linearly as the amount of resource applied increases?
- **Throughput versus resource consumed (i.e., CPU, network I/O, and disk I/O) per transaction.** Does the amount of resource consumed per unit of work increase as scale increases?
- **Latency versus throughput.** Does the latency of a transaction increase as the throughput of a system increases? A system that provides linear throughput scalability might not be useful in the real world if the transaction response times are too long.
- **Statistics.** Measure code path length in both number of instructions and cycles.

OBSERVATION TOOLS ARE THE PRIMARY MEANS TO SCALABLE SOFTWARE

Effective tools are the most significant factor in improving application scalability. Being able to quickly identify a root cause of a scaling issue is paramount. The objective of looking for scaling issues is to easily pinpoint the most significant sources of serialization.

The tools should help identify what type of issue is causing the serialization—the two classic cases being star-

vation resulting from escalating resource requirements as load increases, and increasing idle time as load increases. Ideally, the tools should help identify the source of the scaling issue rather than merely pointing to the object of contention. This helps with identifying not only what the contention point is, but also perhaps some offending code that may be overutilizing a resource. Often, once the source is identified, many obvious optimizations become apparent.

Consider tools that can do the following:

- **Locate key sources of wait time.** What are the contended resources, which one is causing the resource utilization, and how much effect is the contention having on overall performance?
- **Identify hot synchronization locks.** How much wall clock and CPU time is serialized in locking objects, and which code is responsible?
- **Identify nonscalable algorithms.** Which functions or classes become more expensive as the scale of the application increases?
- **Make it clear where the problem lies.** This is done either in the application code, which you can affect, or by pointing to a contention point in a vendor-supplied middleware or operating system. Even though the contention point may lie in a vendor code, it may result from how that code is being called, which can be affected by optimizing the higher-level code.

CMT AND SOFTWARE LICENSING

Another impact of the hardware architecture's scaling characteristics is on software licensing. Application developers often use the number of processors in the system to determine the price of the software. The number of processors has been a convenient measure for software licensing, primarily because of the close correlation between the costs of the hardware platform and the number of processors. By using a license fee indexed by the number of processors, the software vendor can charge a roughly proportional fee for software.

This is, however, based on old assumptions that are no longer true. First of all, an operating system on a CMT platform reports one virtual processor for every thread in the chip, resulting in a very expensive software license for a low-end system. Software vendors have been scrambling to adjust for the latest two-core CMT systems, some opting for one license fee per core, and others for each physical chip. Licensing by core unfairly increases software licenses per dollar unit of hardware.

In the short term, operating system vendors are providing enhancements to report the number of cores

and physical processors in the system, but there is an urgent need for a more appropriate (and fair) solution. It is likely that a throughput-based license fee that uses standard benchmarks will be pursued. This would allow license fees to be charged in accordance with the actual processing power of the platform. Such a scheme would allow software licenses to scale when more advanced virtualization schemes, which divide up processors into subprocessor portions, are used (such as priority-based resource partitioning). These schemes are becoming more commonplace as utility computing and server consolidation become more popular. The opportunity for operating system vendors is to choose a uniform metric that can be measured and reported, based on the actual use by an application.

IMPACT OF CMP FOR OPERATING SYSTEMS

The challenge for the operating system is twofold: providing scalable system services to the applications it hosts, and providing a scalable programming environment that facilitates easy development of parallel programs.

CMP ENHANCEMENTS FOR OPERATING SYSTEMS

An SMP-capable operating system kernel works quite well on CMP hardware. Since each core or hardware thread in a chip has an entire set of registers, they appear to software as individual CPUs. An unchanged operating system will simply implement one logical processor for every hardware thread in the chip. Software threads will be

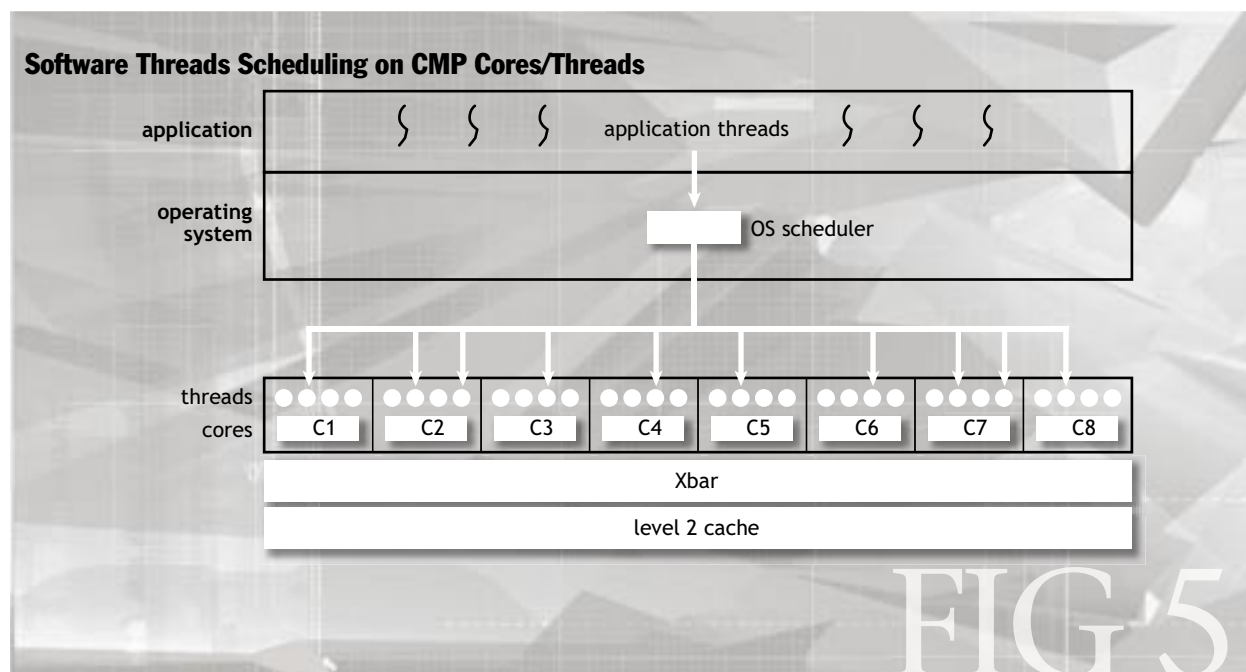
scheduled onto each hardware thread just as in an SMP system, with equal weighting according to the operating system kernel's scheduling policy (see figure 5).

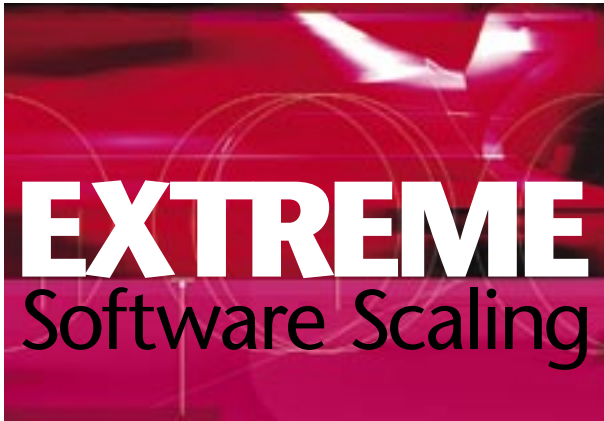
Basic changes to optimize for CMT processors will include elimination of any busy wait loops. For example, the idle loop is typically implemented as a busy spin that checks a run queue looking for more work to do. When multiple hardware threads share a single core, the idle loop running on one thread will have a detrimental effect on other threads sharing the core's pipeline. In this example, leveraging the hardware's ability to park a thread when there is no work to do would be more effective.

Further operating system enhancements will likely be pursued to optimize for the subtle differences of CMPs. For example, with knowledge of the processor architecture and some information about the behavior of the software, the scheduler may be able to optimize the placement of software threads onto specific hardware threads. In the case of a CMP architecture with multiple hardware threads sharing a core, first-level cache, and TLB (translation look-aside buffer), there may be a benefit if software threads with similar memory access patterns (constructive) are colocated on the same core, and those with destructive patterns are separated onto different cores.

OPERATING SYSTEM SCALING

The challenge with scaling operating system services has historically been the shared state between instances of the services. For example, consider a global process table that





needs to be accessed and updated by any program wanting to start a new process. In a multiprocessor system, synchronization techniques must be used to mitigate race conditions when two or more threads attempt to update the process table at the same time.

The common techniques require serialization around either the code that accesses these structures or the data structures themselves. Early attempts to port Unix to SMP hardware were crude—they were typically retrofits of existing operating system codes with simple, coarse-grained serialization. For example, the first SMP Unix systems used a slightly modified implementation with a single global lock around the operating system kernel to serialize all requests to its data structures. Early versions of SunOS (1.x), Linux (2.2), and FreeBSD (4.x) kernels used this approach. Although easy to implement, this approach helps scalability only for applications that seldom use operating system services. Applications that were entirely compute-intensive showed good scalability, but those that used a significant amount of operating system services saw serialization yielding little or no scalability beyond one processor.

In contrast, successful operating system scaling is achieved by minimizing contention, restricting serialization to only fine-grained portions of data structures. In this way, the operating system can execute code within the same region concurrently on multiple processors, serializing only momentarily while accessing shared data structures. This approach does, however, require substantial architectural change to the operating system and in some cases a ground-up redesign focused on scalability.

A well-designed operating system allows high levels of concurrency through its operating system services. In particular, applications invoking system services through libraries, memory allocators, and other system services must be able to execute in parallel even if they access

shared facilities. For example, multiple programs should be able to allocate memory concurrently without serializing. Other areas that are critical to scalability include parallel access to shared hardware (e.g., I/O) and the networking subsystem.

SCALING ENHANCEMENTS IN FREEBSD

FreeBSD has seen a significant amount of scaling effort, starting with 5.x kernels.⁴ Architectural changes include new kernel memory allocators, synchronization routines, the move to ithreads, and the removal of the global kernel lock from activities such as process scheduling, virtual memory, the virtual file system, the UFS (Unix file system), the networking stack, and several common forms of inter-process communication. The scaling work in FreeBSD has successfully improved scaling (estimates suggest to the order of 12 processors).

SCALING ENHANCEMENTS IN LINUX

Scaling was greatly improved in Linux 2.2 kernels by breaking up the global kernel lock. It is said to scale on the order of two to four processors. Linux 2.4 scaling was improved to eight to 16 by introducing much finer-grained locking in the scheduler and I/O subsystem. This improved the scaling of many items, including interrupts and I/O. Later efforts in Linux kernels focused on scaling the scheduler for larger numbers of processes and improving concurrency through the networking subsystem.

SCALING ENHANCEMENTS IN SOLARIS

The Solaris operating system is built around the concept of concurrency, and serialization is restricted to very small and critical parts of data structures. The operating system is designed around the notion that execution contexts are individual software threads, which are scheduled and executed in parallel where possible.

Replacing the original Unix memory allocators with the Slab⁵ and Vmem⁶ allocators led to significant scalability gains. These provide consistent in-time allocations as the object set sizes grow, and they pay special attention to avoid locking by providing per-processor pools of memory that allow allocations and deallocations to occur without having to access global structures.

Scalable I/O is achieved by allowing requesting threads to execute concurrently even within the same device driver, and further by processing interrupts from hardware devices as separate threads, allowing scaling of interrupt handling.⁷

In some cases, there are requirements for high levels of concurrent access to data structures. For example, per-

formance statistics for I/O devices require updates from potentially thousands of concurrent operations. To mitigate contention around these types of structures, statistics are kept on a per-processor basis and then aggregated when required. This allows concurrent access to updates, requiring serialization only when the statistics are read.

The Solaris networking code was rearchitected to eliminate the majority of the global data structures by introducing a per-connection vertical perimeter.⁸ This allows the TCP/IP implementation to operate in near-lockless mode within a single connection, requiring locking only when global events such as routing changes occur.

Integrated observation tools are key to optimizing scaling issues. Facilities for observing sources of locking contention on systems with live workloads have been critical to making improvements in important areas. More recently, Dtrace, perhaps one of the more revolutionary approaches to performance optimization, allows dynamic instrumentation of C and Java code.⁹ It can quickly pinpoint sources of contention from the top of the application stack through the operating system.

These types of techniques allow the Solaris kernel to scale to thousands of threads, up to 1 million I/Os per second, and several hundred physical processors. Conveniently, this scaling work can be leveraged for CMP systems. Techniques such as those described here, which are vital for large SMP scaling, are now required even for entry-level CMP systems. Within the next five years, expect to see CMP hardware scaling to as many as 512 processor threads per system, pushing the requirements of operating system scaling past the extreme end of that realized today.

OPERATING SYSTEM UTILIZATION METRICS

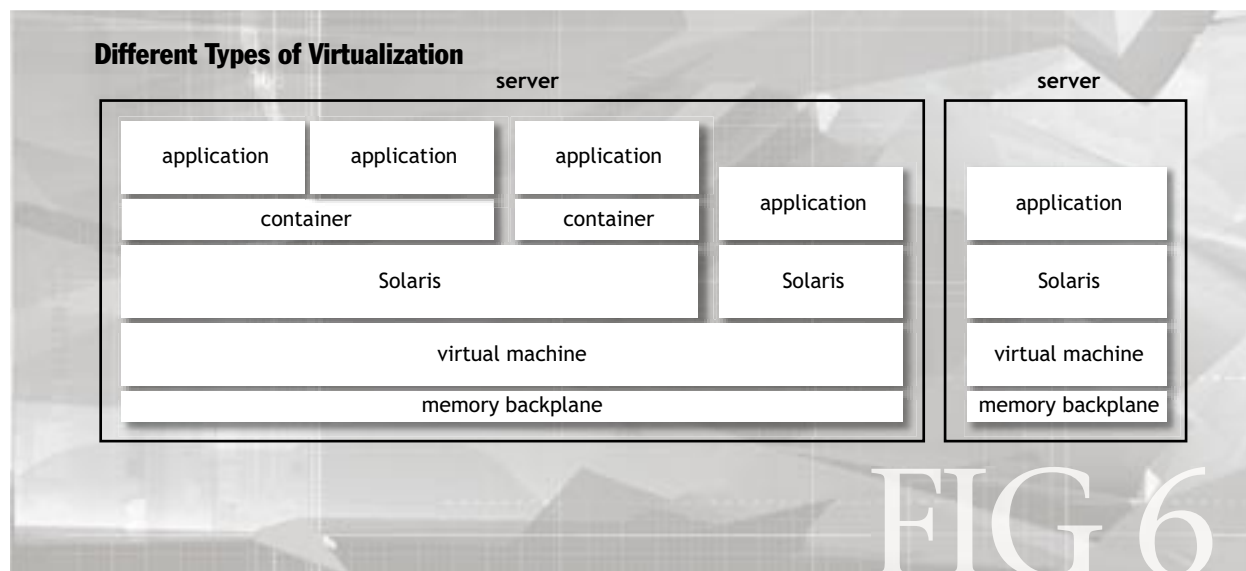
The reporting of processor utilization on systems with multithreaded cores poses a challenge. In a single-core chip, throughput often increases proportionally with processor utilization. In a multithreaded chip, there is much greater opportunity for sharing of resources between hardware threads, and therefore a nonlinear relationship exists between throughput and the actual utilization of a processor. As a result, calculation of “headroom” based on reported processor utilization may no longer be accurate.

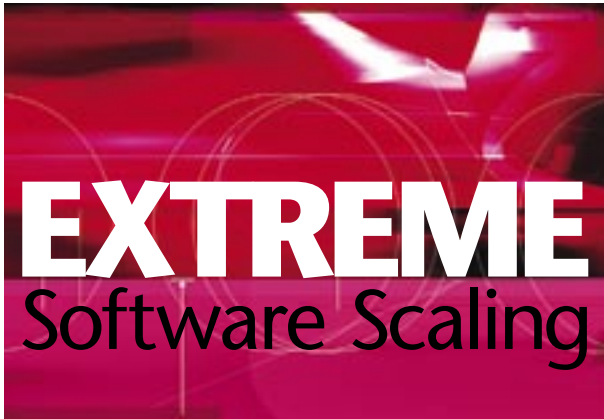
For example, a processor core with two threads (such as an Intel Xeon) presents itself to the operating system as two separate processors. If a software thread fully uses one of the threads and the other is completely idle, the processor will appear 50 percent busy and be reported as such by the operating system. Running two of these threads on the processor may often yield only a 10 percent throughput increase on Xeon architecture, but since both threads are utilized, it will report as 100 percent busy. So this system now reports 50 percent utilization when it’s at 90 percent of its maximum throughput.

This effect will vary depending on how many of the resources are shared by hardware threads within the processor, and ultimately will need some redefinition of the meaning of system utilization metrics, together with some new facilities for reporting. The impact on capacity planning methodology will also need to be considered.

LEVERAGING VIRTUALIZATION FOR PARALLELISM

So far we have examined how to find ways to use the many hardware threads available with CMTs by scaling individual applications or operating systems. Another





way to use these resources effectively is to run multiple nonscalable applications or even several unoptimized operating systems at once, using techniques such as operating system or server virtualization.

These facilities typically allow multiple instances of an application to be consolidated onto a single server (see figure 6).

For example, the Solaris Container facility allows multiple applications to reside within a single operating system instance. In such an environment, you can leverage the cumulative concurrency as applications are added. By adding two Web servers, each of which has concurrency of 16 threads, you can potentially increase the system-wide concurrency to 32 threads. This side effect presents a useful mechanism that allows you to deploy applications with limited scalability in a manner that can exploit the full concurrency of a CMP system.

Another relevant virtualization technology is the virtual machine environment, which allows multiple operating system instances to run on a single hardware platform. Examples of virtual machine technologies are VMware and Xen. These environments allow consolidation of applications and operating systems on a single system, which provides a mechanism to deploy even nonscalable operating systems on CMP architectures, albeit with a little more complexity.

CMP REQUIRES A RETHINKING BY DEVELOPERS

The introduction of CMP systems represents a significant opportunity to scale systems in a new dimension. The most significant impact of CMP systems is that the degree of scaling is being increased by an order of magnitude: what was a low-end one- to two-processor entry-level system should now be viewed as a 16- to 32-way system, and soon even midrange systems will be scaling to several hundred ways.

For application developers, this represents a new or revised focus on intra-machine scalability within applications and a rethinking of how software license fees are calculated. For operating system developers, scalability to hundreds of ways is going to be a requirement. For deployment practitioners, CMP represents a new way to scale applications and will require consideration in the systems we architect, the way we tune, and the techniques we use for capacity planning. Q

REFERENCES

1. AMD Opteron Processor; <http://www.amd.com>.
2. Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro* 25 (2): 21–29.
3. Amdahl, G. M. 1967. Validity of the single-processor approach to achieving large-scale computing capabilities. *Proceedings of AFIPS Conference*: 483–485.
4. The FreeBSD SMP Project; <http://www.freebsd.org/smp/>.
5. Bonwick, J. 1994. The Slab allocator: an object-caching kernel memory allocator. Sun Microsystems.
6. Bonwick, J., and Adams, J. 2001. Magazines and Vmem: extending the Slab allocator to many CPUs and arbitrary resources. Sun Microsystems and California Institute of Technology.
7. Kleiman, S., and Eykholt, J. 1995. Interrupts as threads. *ACM Sigops Operating Systems Review* 29 (2): 21–26.
8. Tripathi, S. 2005. Solaris OS network performance. Sun White Paper (February).
9. Cantrill, B. M., Shapiro, M. W., Leventhal, A.H. 2004. Dynamic instrumentation of production systems. *Unix Proceedings*.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RICHARD McDougall, had he lived 100 years ago, would have had the hood open on the first four-stroke internal combustion gasoline-powered vehicle, exploring new techniques for making improvements. He would be looking for simple ways to solve complex problems and helping pioneering owners understand how the technology worked to get the most from their new experience. These days, McDougall uses technology to satisfy his curiosity. He is a Distinguished Engineer at Sun Microsystems, specializing in operating systems technology and system performance. McDougall is the author of *Solaris Internals* (Prentice Hall, 2000; second edition, 2005), and *Resource Management* (Prentice Hall, 1999). © 2005 ACM 1542-7730/05/0900 \$5.00

DEVELOPMENT LIFECYCLE PRACTICES



Managing
Projects & Teams



Plan-Driven
Development



Agile
Development



Process Improvement
& Measurement



Testing & Quality
Assurance



Security &
Special Topics

BETTER SOFTWARE 05 **CONFERENCE & EXPO**

THE LATEST IN SOFTWARE
DEVELOPMENT TODAY

► *Find the Balance*



www.sqe.com

**SEPTEMBER 19-22, 2005
SAN FRANCISCO, CA**
Hyatt Regency San Francisco Airport

REGISTER NOW!

www.sqe.com/bettersoftwareconf



The Price of

In the late 1990s, our research group at DEC was one of a growing number of teams advocating the CMP (chip multiprocessor) as an alternative to highly complex single-threaded CPUs. We were designing the Piranha system,¹ which was a radical point in the CMP design space in that we used very simple cores (similar to the early RISC designs of the late '80s) to provide a higher level of thread-level parallelism. Our main goal was to achieve the best commercial workload performance for a given silicon budget.

Today, in developing Google's computing infrastructure, our focus is broader than performance alone.

The merits of a particular architecture are measured by

answering the following question: Are

you able to afford the computational

capacity you need? The high-compu-

tational demands that are inherent in

most of Google's services have led us

to develop a deep understanding of the

overall cost of computing, and continu-

ally to look for hardware/software designs that optimize

performance per unit of cost.

This article addresses some of the cost trends in a large-scale Internet service infrastructure and highlights

the challenges and opportunities for CMP-based systems

to improve overall computing platform cost efficiency.

Performance

LUIZ ANDRÉ BARROSO, GOOGLE

An Economic Case for Chip Multiprocessing

UNDERSTANDING SYSTEM COST

The systems community has developed an arsenal of tools to measure, model, predict, and optimize performance.

The community's appreciation and understanding of cost factors, however, remain less developed. Without

thorough consideration and understanding of cost, the true merits of any one technology or product remain

unproven.

We can break down the TCO (total cost of ownership) of a large-scale computing cluster into four main compo-

nents: price of the hardware, power (recurring and initial data-center investment), recurring data-center operations

costs, and cost of the software infrastructure.

Often the major component of TCO for commercial deployments is software. A cursory inspection of the price

breakdown for systems used in TPC-C benchmark filings shows that per-CPU costs of just operating systems and

database engines can range from \$4,000 to \$20,000.²

Once the license fees for other system software compo-

nents, applications, and management software are added up, they can dwarf all other components of cost. This is

especially true for deployments using mid- and low-end servers, since those tend to have larger numbers of less

The Price of Performance

expensive machines but can incur significant software costs because of still-commonplace per-CPU or per-server license-fee policies.

Google's choice to produce its own software infrastructure in-house and to work with the open source community changes that cost distribution by greatly reducing software costs (software development costs still exist, but are amortized over large CPU deployments). As a result, it needs to pay special attention to the remaining components of cost. Here I will focus on cost components that are more directly affected by system-design choice: hardware and power costs.

Figure 1 shows performance, performance-per-server price, and performance-per-watt trends from three successive generations of Google server platforms. Google's hardware solutions include the use of low-end servers.³ Such systems are based on high-volume, PC-class components and thus deliver increasing performance for roughly the same cost over successive generations, resulting in the upward trend of the performance-per-server price curve. Google's fault-tolerant software design methodology enables it to deliver highly available services based on these relatively less-reliable building blocks.

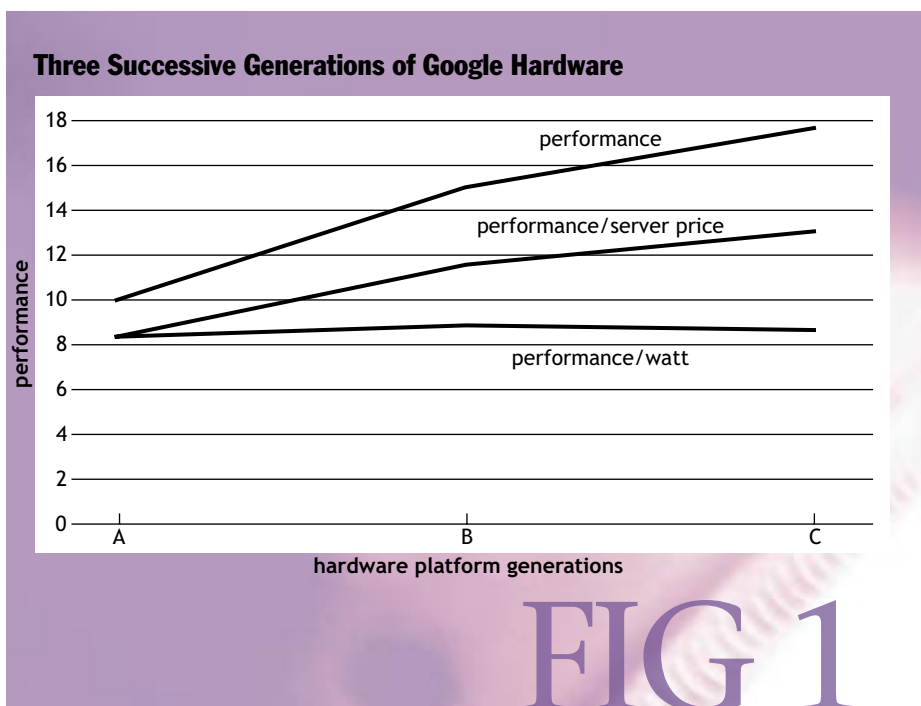
Nevertheless, performance per watt has remained roughly flat over time, even after significant efforts to design for power efficiency. In other words, every gain in performance has been accompanied by a proportional inflation in overall platform power consumption. The result of these trends is that power-related costs are an increasing fraction of the TCO.

Such trends could have a significant impact on how computing costs are factored. The following analysis ignores other indirect power costs and focuses solely on the cost of energy. A typical low-end x86-based server today can cost about \$3,000 and consume an average of 200 watts (peak consumption

can reach over 300 watts). Typical power delivery inefficiencies and cooling overheads will easily double that energy budget. If we assume a base energy cost of nine cents per kilowatt hour and a four-year server lifecycle, the energy costs of that system today would already be more than 40 percent of the hardware costs.

And it gets worse. If performance per watt is to remain constant over the next few years, power costs could easily overtake hardware costs, possibly by a large margin. Figure 2 depicts this extrapolation assuming four different annual rates of performance and power growth. For the most aggressive scenario (50 percent annual growth rates), power costs by the end of the decade would dwarf server prices (note that this doesn't account for the likely increases in energy costs over the next few years). In this extreme situation, in which keeping machines powered up costs significantly more than the machines themselves, one could envision bizarre business models in which the power company will provide you with free hardware if you sign a long-term power contract.

The possibility of computer equipment power consumption spiraling out of control could have serious consequences for the overall affordability of computing, not to mention the overall health of the planet. It should be noted that although the CPUs are responsible for only



a fraction of the total system power budget, that fraction can easily reach 50 percent to 60 percent in low-end server platforms.

THE CMP AND COMPUTING EFFICIENCY

The eventual introduction of processors with CMP technology is the best (and perhaps only) chance to avoid the dire future envisioned above. As discussed in the opening article of this issue ("The Future of Microprocessors," by Kunle Olukotun and Lance Hammond), if thread-level parallelism is available, using the transistor and energy budget for additional cores is more likely to yield higher performance than any other techniques we are aware of. In such a thread-rich environment, prediction and speculation techniques need to be extremely accurate to justify the extra energy and real estate they require, as there will be nonspeculative instructions ready to execute from other threads. Unfortunately, many server-class workloads are known to exhibit poor instruction-level parallelism;⁴ therefore, they are a poor match for the aggressive speculative out-of-order cores that are common today.

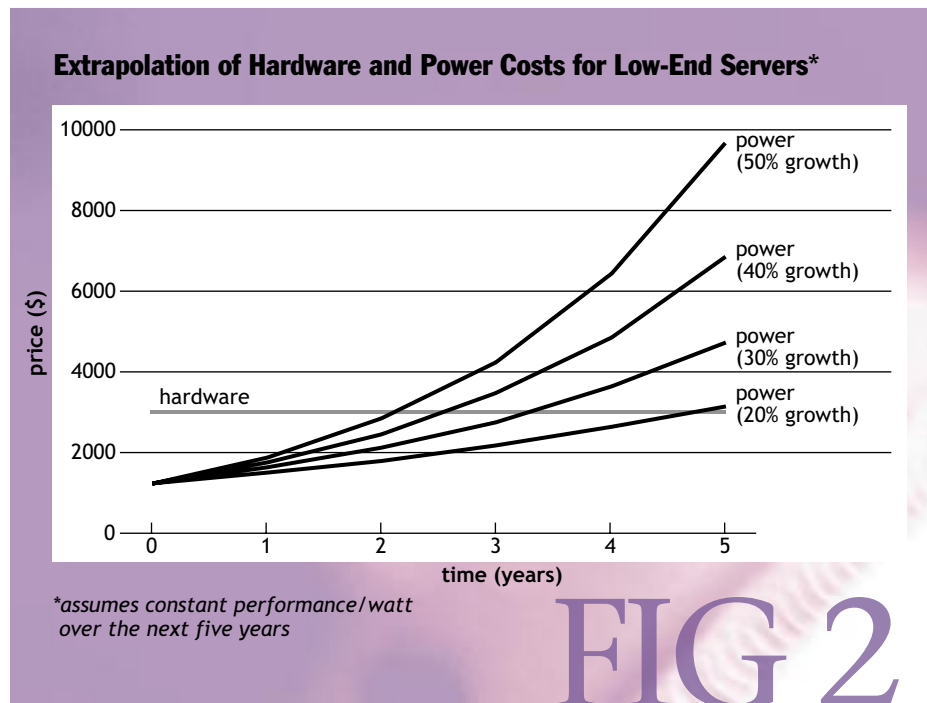
Some key workloads at Google share such behavior. Our index-serving application, for example, retires on average only one instruction every two CPU cycles on modern processors, badly underutilizing the multiple issue slots and functional units available. This is caused by the use of data structures that are too large for on-chip caches, and a data-dependent control flow that exposes

the pipeline to large DRAM latencies. Such behavior also causes the memory system to be underutilized, since often a new memory access cannot be issued until the result of a previous one is available. There is enough unpredictability in both control flow and memory access streams to render speculation techniques relatively ineffective. This same workload, however, exhibits excellent thread-level speedup on traditional multiprocessors, simultaneous multithreaded systems, and CMPs.⁵

The Piranha implementation took the lessons from commercial workload behavior to heart: If there are enough threads (hardware and software), one should never have to speculate. The eight CPU cores were a throwback to early RISC designs: single-issue, in-order, nonspeculative. The first Piranha chip was expected to outperform state-of-the-art CPUs by more than a factor of two at nearly half the power consumption. What makes this especially significant is that this was achieved despite our team having completely ignored power efficiency as a design target. This is a good illustration of the inherent power-efficiency advantages of the CMP model.

Recent product announcements also provide insights into the power-efficiency potential of CMP microarchitectures. Both AMD and Intel are introducing CMP designs that stay within approximately the same power envelope of their previous-generation single-core offerings. For example, AMD reports that its dual-core Opteron 275 model outperforms its single-core equivalent (Opteron

248) by about 1.8 times on a series of benchmarks,⁶ at a power envelope increase of less than 7 percent. Even if we pessimistically assume that the whole platform power increases by that same amount, the power efficiency of the dual-core platform (performance per watt) is still nearly 70 percent better than the single-core platform. Indeed, process technology improvements do play a large role in achieving this, but the fact remains that for the first time in many processor generations we are looking at dramatic power-efficiency improvements.



The Price of Performance

SLOW PACE

In our first Piranha paper published in 2000 we described chip multiprocessing as an inevitable next step in micro-architectural evolution. Although this is no longer a controversial view, it is nevertheless surprising that it has taken so long for this architecture to achieve widespread acceptance. I am particularly surprised that more aggressive CMP architectures—those (like Piranha) that trade single-threaded performance for additional thread-level parallelism—are only now beginning to appear in commercial products⁷ and are unlikely to be widely available for quite some time.

The commercial introduction of CMPs seems to be following a more measured approach in which fairly complex cores are being slowly added to the die as the transistor budget increases every process generation. If CMPs have such compelling potential, why is it taking so long for that potential to be realized? There are four main reasons for this:

It's the power envelope, stupid. As it turned out, contrary to what we envisioned during the Piranha development, design complexity and performance alone were not compelling enough to trigger a switch to CMP architectures; power was. In order to steer away from expensive cooling technologies, chip developers had to stay within power density boundaries that became increasingly difficult to meet with conventional techniques.

Marketing matters. Megahertz is a performance metric that is easy to understand and communicate to consumers. Although it is a very poor indicator of application performance, the same can be said for most popular benchmarks. When given a choice between a bogus metric that sells and one that doesn't, the outcome is predictable. Unfortunately, the MHz competition has reinforced the direction toward larger and more complex single-threaded systems, and away from CMPs.

Execution matters. Many of us underestimated the incredible engineering effort that went into making conventional complex cores into very successful products. Seemingly suboptimal architectures can be made into winning solutions with the right combination of talent, drive, and execution.

Threads aren't everywhere yet. Although server-class workloads have been multithreaded for years, the same cannot be said yet for desktop workloads. Since desktop volume still largely subsidizes the enormous cost of server CPU development and fabrication, the lack of

threads in the desktop has made CMPs less universally compelling. I will expand on this issue later in this article.

DREADING THREADING

Much of the industry's slowness in adopting CMP designs reflects a fear that the CMP opportunity depends on having enough threads to take advantage of that opportunity. Such fear seems to be based mainly on two factors: parallel programming complexity and the thread-level speedup potential of common applications.

The complexity of parallel software can slow down programmer productivity by making it more difficult to write correct and efficient programs. Computer science students' limited exposure to parallel programming, lack of popular languages with native support for parallelism, and the slow progress of automatic compiler parallelization technology all contribute to the fear that many applications will not be ready to take advantage of multithreaded chips.

There is reason for optimism, though. The ever-growing popularity of small multiprocessors is exposing more programmers to parallel hardware. More tools to spot correctness and performance problems are becoming available (e.g., thread checkers⁸ and performance debuggers⁹). Also, a few expert programmers can write efficient threaded code that is in turn leveraged by many others. Fast-locking and thread-efficient memory allocation libraries are good examples of programming work that is highly leveraged. On a larger scale, libraries such as Google's MapReduce¹⁰ can make it easier for programmers to write efficient applications that mine huge datasets using hundreds or thousands of threads.

While it's true that some algorithms are hard to parallelize efficiently, the majority of problem classes that demand the additional performance of CMPs are not. The general principle here is that, with few exceptions, the more data one has, the easier it is to obtain parallel speedup. That's one of the reasons why database applications have been run as parallel workloads successfully for well over a decade. At Google we have generally been able to tune our CPU-intensive workloads to scale to increasing numbers of hardware threads whenever needed—that is, whenever servers with higher numbers of hardware contexts become economically attractive.

The real challenge for CMPs is not at the server but the desktop level. Many popular desktop applications have not been parallelized yet, in part because they manipulate

modest datasets, and in part because multithreaded CPUs have only recently been introduced to that market segment. As more data-intensive workloads (such as speech recognition) become common at the desktop, CMP systems will become increasingly attractive for that segment.

It is important to note that CMPs are a friendly target platform for applications that don't parallelize well. Communication between concurrent threads in a CMP can be an order of magnitude faster than in traditional SMP systems, especially when using shared on-chip caches. Therefore, workloads that require significant communication or synchronization among threads will pay a smaller performance penalty. This characteristic of CMP architectures should ease the programming burden involved in initial parallelization of the established code base.

CMP HEADING FOR MAINSTREAM ACCEPTANCE

A highly cost-efficient distributed computing system is essential to large-scale services such as those offered by Google. For these systems, given the distributed nature of the workloads, single-threaded performance is much less important than the aggregate cost/performance ratio of an entire system. Chip multiprocessing is a good match for such requirements. When running these inherently parallel workloads, CMPs can better utilize on-chip resources and the memory system than traditional wide-issue single-core architectures, leading to higher performance for a given silicon budget. CMPs are also fundamentally more power-efficient than traditional CPU designs and therefore will help keep power costs under control over the next few years. Note, however, that CMPs cannot solve the power-efficiency challenge alone, but can simply mitigate it for the next two or three CPU generations. Fundamental circuit and architectural innovations are still needed to address the longer-term trends.

The computing industry is ready to embrace chip multiprocessing as the mainstream solution for the desktop and server markets, yet it appears to be doing so with some reluctance. CMP parallelism is being introduced only when it is absolutely necessary to remain within a safe thermal envelope. This approach minimizes any significant losses in single-threaded performance, but it is unlikely to realize the full cost-efficiency potential of chip multiprocessing. A riskier bet on slower cores could have a much larger positive impact on the affordability of high-performance systems. Q

REFERENCES

1. Barroso, L. A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R.,

and Verghese, B. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. *Proceedings of the 27th ACM International Symposium on Computer Architecture* (June), Vancouver, BC.

2. Transaction Processing Performance Council. Executive summary reports for TPC-C benchmark filings; <http://www.tpc.org>.
3. Hoelzle, U., Dean, J., and Barroso, L. A. 2003. Web search for a planet: the architecture of the Google cluster. *IEEE Micro Magazine* (April).
4. Ranganathan, P., Gharachorloo, K., Adve, S., and Barroso, L.A. 1998. Performance of database workloads on shared memory systems with out-of-order processors. *Proceedings of the Eighth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, CA.
5. See Reference 3.
6. AMD competitive server benchmarks; http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_8800~97051,00.html.
7. Kongetira, P., Aingaran, K., and Olukotun, K. 2005. Niagara: a 32-way multithreaded SPARC processor. *IEEE Micro Magazine* (March/April); <http://www.computer.org/micro>.
8. Intel Corporation. Intel thread checker; <http://developer.intel.com/software/products/threading/tcwin>.
9. Seward, J. Valgrind; <http://valgrind.kde.org/>.
10. Dean, J., and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. *Proceedings of OSDI*, San Francisco, CA.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ACKNOWLEDGMENTS

The author thanks Wolf-Dietrich Weber and Christopher Lyle Johnson for their careful review of the manuscript.

LUIZ ANDRÉ BARROSO is a principal engineer at Google, where he leads the platforms engineering group. He has worked on several aspects of Google's systems infrastructure, including load balancing, fault detection and recovery, communication libraries, performance optimization, and the computing platform design. Prior to Google he was on the research staff at Compaq and DEC, where he investigated processor and memory system architectures for commercial workloads and co-architected the Piranha system. Barroso holds a Ph.D. in computer engineering from USC, and a B.Sc. and M.S. in electrical engineering from PUC-Rio, Brazil. © 2005 ACM 1542-7730/05/0900 \$5.00

Leveraging the full power
of multicore processors demands
new tools and new thinking
from the software industry.

Software and the Concurrency Revolution

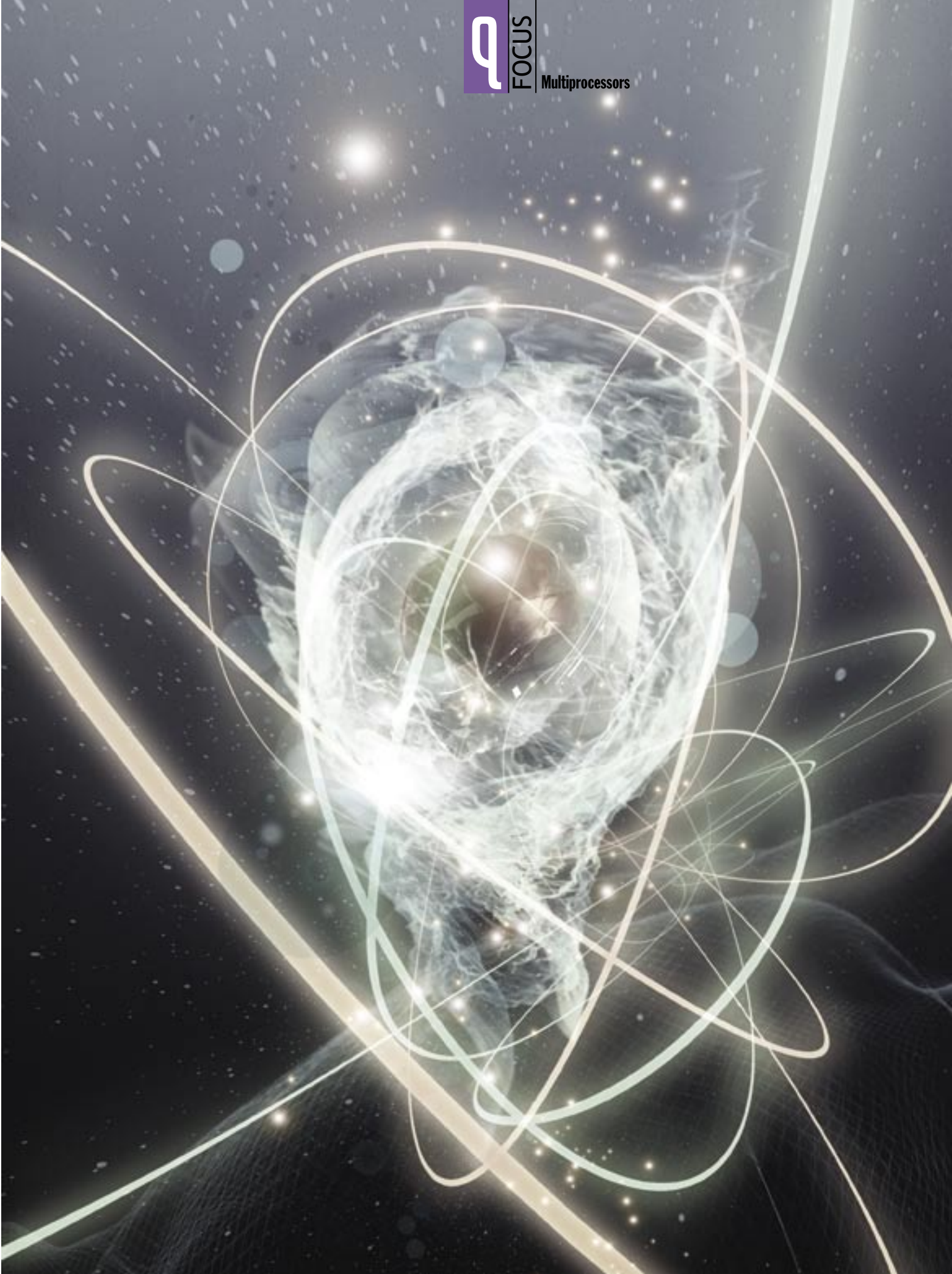
Concurrency has long been touted as the “next big thing” and “the way of the future,” but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software.

The introductory article in this issue (“The Future of Microprocessors” by Kunle Olukotun and Lance Hammond) describes the hardware imperatives behind this shift in computer architecture from uniprocessors to multicore processors, also known as CMPs (chip multiprocessors). (For related analysis, see “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.”¹)

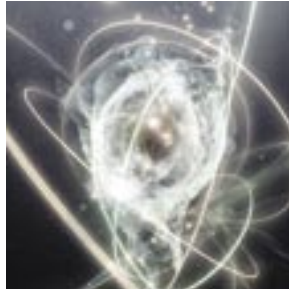
In this article we focus on the implications of concurrency for software and its consequences for both programming languages and programmers.

The hardware changes that Olukotun and Hammond describe represent a fundamental shift in computing. For the past three decades, improvements in semiconductor fabrication and processor implementation produced steady increases in the speed at which computers executed existing sequential programs. The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. For the foreseeable future, today’s desktop applications will

HERB SUTTER AND JAMES LARUS, MICROSOFT



Software and the Concurrency Revolution



not run much faster than they do now. In fact, they may run slightly slower on newer chips, as individual cores become simpler and run at lower clock speeds to reduce power consumption on dense multicore processors.

That brings us to a fundamental turning point in software development, at least for mainstream software. Computers will continue to become more and more capable, but programs can no longer simply ride the hardware wave of increasing performance unless they are highly concurrent.

Although multicore performance is the forcing function, we have other reasons to want concurrency: notably, to improve responsiveness by performing work asynchronously instead of synchronously. For example, today's applications must move work off the GUI thread so it can redraw the screen while a computation runs in the background.

But concurrency is hard. Not only are today's languages and tools inadequate to transform applications into parallel programs, but also it is difficult to find parallelism in mainstream applications, and—worst of all—concurrency requires programmers to think in a way humans find difficult.

Nevertheless, multicore machines are the future, and we must figure out how to program them. The rest of this article delves into some of the reasons why it is hard, and some possible directions for solutions.

CONSEQUENCES: A NEW ERA IN SOFTWARE

Today's concurrent programming languages and tools are at a level comparable to sequential programming at the beginning of the structured programming era. Semaphores and coroutines are the assembler of concurrency, and locks and threads are the slightly higher-level structured constructs of concurrency. What we need is OO for concurrency—higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs.

For several reasons, the concurrency revolution is likely to be more disruptive than the OO revolution.

First, concurrency will be integral to higher performance. Languages such as C ignored OO and remained usable for many programs. If concurrency becomes the sole path to higher-performance hardware, commercial and systems programming languages will be valued on their support for concurrent programming. Existing languages, such as C, will gain concurrent features beyond simple models such as pthreads. Languages that fail to support concurrent programming will gradually die away and remain useful only when modern hardware is unimportant.

The second reason that concurrency will be more disruptive than OO is that, although sequential programming is hard, concurrent programming is demonstrably more difficult. For example, context-sensitive analysis of sequential programs is a fundamental technique for taking calling contexts into account when analyzing a program. Concurrent programs also require synchronization analysis, but simultaneously performing both analyses is provably undecidable.²

Finally, humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among simple collections of partially ordered operations.

DIFFERENCES BETWEEN CLIENT AND SERVER APPLICATIONS

Concurrency is a challenging issue for client-side applications. For many server-based programs, however, concurrency is a “solved problem,” in that we routinely architect concurrent solutions that work well, although programming them and ensuring they scale can still require a huge effort. These applications typically have an abundance of parallelism, as they simultaneously handle many independent request streams. For example, a Web server or Web site independently executes thousands of copies of the same code on mostly nonoverlapping data.

In addition, these executions are well isolated and share state via an abstract data store, such as a database that supports highly concurrent access to structured data. The net effect is that code that shares data through a database can keep its “peaceful easy feeling”—the illusion of living in a tidy, single-threaded universe.

The world of client applications is not nearly as well structured and regular. A typical client application executes a relatively small computation on behalf of a single user, so concurrency is found by dividing a computation into finer pieces. These pieces, say the user interface and program's computation, interact and share data in myriad ways. What makes this type of program difficult

to execute concurrently are nonhomogeneous code; fine-grained, complicated interactions; and pointer-based data structures.

PROGRAMMING MODELS

Today, you can express parallelism in a number of different ways, each applicable to only a subset of programs. In many cases, it is difficult, without careful design and analysis, to know in advance which model is appropriate for a particular problem, and it is always tricky to combine several models when a given application does not fit cleanly into a single paradigm.

These parallel programming models differ significantly in two dimensions: the granularity of the parallel operations and the degree of coupling between these tasks. Different points in this space favor different programming models, so let's examine these axes in turn.

Operations executed in parallel can range from single instructions, such as addition or multiplication, to complex programs that take hours or days to run. Obviously, for small operations, the overhead costs of the parallel infrastructure are significant; for example, parallel instruction execution generally requires hardware support. Multicore processors reduce communication and synchronization costs, as compared with conventional multiprocessors, which can reduce the overhead burden on smaller pieces of code. Still, in general, the finer grained the task, the more attention must be paid to the cost of spawning it as a separate task and providing its communication and synchronization.

The other dimension is the degree of coupling in the communication and synchronization between the operations. The ideal is none: operations run entirely independently and produce distinct outputs. In this case, the operations can run in any order, incur no synchronization or communications costs, and are easily programmed without the possibility of data races. This state of affairs is rare, as most concurrent programs share data among their operations. The complexity of ensuring correct and efficient operation increases as the operations become more diverse. The easiest case is executing the same code for each operation. This type of sharing is often regular and can be understood by analyzing only a single task. More challenging is irregular parallelism, in which the operations are distinct and the sharing patterns are more difficult to comprehend.

INDEPENDENT PARALLELISM

Perhaps the simplest and best-behaved model is independent parallelism (sometimes called “embarrassingly paral-

lel tasks”), in which one or more operations are applied independently to each item in a data collection.

Fine-grained data parallelism relies on the independence of the operations executed concurrently. They should not share input data or results and should be executable without coordination. For example:

```
double A[100][100];
...
A = A * 2;
```

multiplies each element of a 100x100 array by 2 and stores the result in the same array location. Each of the 10,000 multiplications proceeds independently and without coordination with its peers. This is probably more concurrency than necessary for most computers, and its granularity is very fine, so a more practical approach would partition the matrix into $n \times m$ blocks and execute the operations on the blocks concurrently.

At the other end of the granularity axis, some applications, such as search engines, share only a large read-only database, so concurrently processing queries requires no coordination. Similarly, large simulations, which require many runs to explore a large space of input parameters, are another embarrassingly parallel application.

REGULAR PARALLELISM

The next step beyond independent parallelism is to apply the same operation to a collection of data when the computations are mutually dependent. An operation on one piece of data is dependent on another operation if there is communication or synchronization between the two operations.

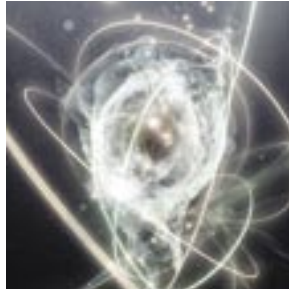
For example, consider a stencil computation that replaces each point in an array, the average of its four nearest neighbors:

$$A[i, j] = (A[i-1, j] + A[i, j-1] + A[i+1, j] + A[i, j+1]) / 4;$$

This computation requires careful coordination to ensure that an array location is read by its neighbors before being replaced by its average. If space is no concern, then the averages can be computed into a new array. In general, other more structured computation strategies, such as traversing the array in a diagonal wavefront, will produce the same result, with better cache locality and lower memory consumption.

Regular parallel programs may require synchronization or carefully orchestrated execution strategies to produce the correct results, but unlike general parallelism, the

Software and the Concurrency Revolution



code behind the operations can be analyzed to determine how to execute them concurrently and what data they share. This advantage is sometimes hypothetical, since program analysis is an imprecise discipline, and sufficiently complex programs are impossible for compilers to understand and restructure.

At the other end of the granularity axis, computations on a Web site are typically independent except for accesses to a common database. The computations run in parallel without a significant amount of coordination beyond the database transactions. This ensures that concurrent access to the same data is consistently resolved.

UNSTRUCTURED PARALLELISM

The most general, and least disciplined, form of parallelism is when the concurrent computations differ, so that their data accesses are not predictable and need to be coordinated through explicit synchronization. This is the form of parallelism most common in programs written using threads and explicit synchronization, in which each thread has a distinct role in the program. In general, it is difficult to say anything specific about this form of parallelism, except that conflicting data accesses in two threads need explicit synchronization; otherwise, the program will be nondeterministic.

THE PROBLEM OF SHARED STATE, AND WHY LOCKS AREN'T THE ANSWER

Another challenging aspect of unstructured parallelism is sharing unstructured state. A client application typically manipulates shared memory organized as unpredictably interconnected graphs of objects.

When two tasks try to access the same object, and one could modify its state, if we do nothing to coordinate the tasks, we have a data race. Races are bad, because the concurrent tasks can read and write inconsistent or corrupted values.

There are a rich variety of synchronization devices that can prevent races. The simplest of these is a lock. Each task that wants to access a piece of shared data must

acquire the lock for that data, perform its computation, and then release the lock so other operations on the data can proceed. Unfortunately, although locks work, they pose serious problems for modern software development.

A fundamental problem with locks is that they are not composable. You can't take two correct lock-based pieces of code, combine them, and know that the result is still correct. Modern software development relies on the ability to compose libraries into larger programs, and so it is a serious difficulty that we cannot build on lock-based components without examining their implementations.

The composability issue arises primarily from the possibility of deadlock. In its simplest form, deadlock happens when two locks might be acquired by two tasks in opposite order: task T1 takes lock L1, task T2 takes lock L2, and then T1 tries to take L2 while T2 tries to take L1. Both block forever. Because this can happen any time two locks can be taken in opposite order, calling into code you don't control while holding a lock is a recipe for deadlock.

That is exactly what extensible frameworks do, however, as they call virtual functions while holding a lock. Today's best-of-breed commercial application frameworks all do this, including the .NET Frameworks and the Java standard libraries. We have gotten away with it because developers aren't yet writing lots of heavily concurrent programs that do frequent locking. Many complex models attempt to deal with the deadlock problem—with backoff-and-retry protocols, for example—but they require strict discipline by programmers, and some introduce their own problems (e.g., livelock).

Techniques for avoiding deadlock by guaranteeing locks will always be acquired in a safe order do not compose, either. For example, lock leveling and lock hierarchies prevent programs from acquiring locks in conflicting order by requiring that all locks at a given level be acquired at once in a predetermined order, and that while holding locks at one level, you can acquire additional locks only at higher levels. Such techniques work inside a module or framework maintained by a team (although they're underused in practice), but they assume control of an entire code base. That severely restricts their use in extensible frameworks, add-in systems, and other situations that bring together code written by different parties.

A more basic problem with locks is that they rely on programmers to strictly follow conventions. The relationship between a lock and the data that it protects is implicit, and it is preserved only through programmer discipline. A programmer must always remember to take the right lock at the right point before touching shared

data. Conventions governing locks in a program are sometimes written down, but they're almost never stated precisely enough for a tool to check them.

Locks have other more subtle problems. Locking is a global program property, which is difficult to localize to a single procedure, class, or framework. All code that accesses a piece of shared state must know and obey the locking convention, regardless of who wrote the code or where it resides.

Attempts to make synchronization a local property do not work all the time. Consider a popular solution such as Java's `synchronized` methods. Each of an object's methods can take a lock on the object, so no two threads can directly manipulate the object's state simultaneously. As long as an object's state is accessed only by its methods and programmers remember to add the `synchronized` declaration, this approach works.

There are at least three major problems with `synchronized` methods. First, they are not appropriate for types whose methods call virtual functions on other objects (e.g., Java's `Vector` and .NET's `SyncHashTable`), because calling into third-party code while holding a lock opens the possibility of deadlock. Second, `synchronized` methods can perform too much locking, by acquiring and releasing locks on all object instances, even those never shared across threads (typically the majority). Third, `synchronized` methods can also perform too little locking, by not preserving atomicity when a program calls multiple methods on an object or on different objects. As a simple example of the latter, consider a banking transfer:

```
account1.Credit(amount); account2.Debit(amount)
```

Per-object locking protects each call, but does not prevent another thread from seeing the inconsistent state of the two accounts between the calls. Operations of this type, whose atomicity does not correspond to a method call boundary, require additional, explicit synchronization.

LOCK ALTERNATIVES

For completeness, we note two major alternatives to locks. The first is *lock-free programming*. By relying on a deep knowledge of a processor's memory model, it is possible to create data structures that can be shared without explicit locking. Lock-free programming is difficult and fragile; inventing a new lock-free data-structure implementation is still often a publishable result.

The second alternative is *transactional memory*, which brings the central idea of transactions from databases into programming languages. Programmers write their

programs as a series of explicitly atomic blocks, which appear to execute indivisibly, so concurrently executing operations see the shared state strictly before or after an atomic action executes. Although many people view transactional memory as a promising direction, it is still a subject of active research.

WHAT WE NEED IN PROGRAMMING LANGUAGES

We need higher-level language abstractions, including evolutionary extensions to current imperative languages, so that existing applications can incrementally become concurrent. The programming model must make concurrency easy to understand and reason about, not only during initial development but also during maintenance.

EXPLICIT, IMPLICIT, AND AUTOMATIC PARALLELIZATION

Explicit programming models provide abstractions that require programmers to state exactly where concurrency can occur. The major advantage of expressing concurrency explicitly is that it allows programmers to take full advantage of their application domain knowledge and express the full potential concurrency in the application. It has drawbacks, however. It requires new higher-level programming abstractions and a higher level of programmer proficiency in the presence of shared data.

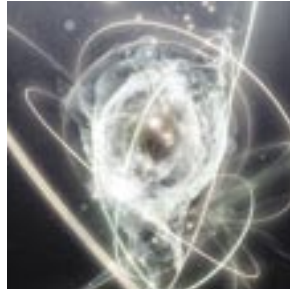
Implicit programming models hide concurrency inside libraries or behind APIs, so that a caller retains a sequential worldview while the library performs the work in parallel. This approach lets naïve programmers safely use concurrency. Its main drawback is that some kinds of concurrency-related performance gains can't be realized this way. Also, it is difficult to design interfaces that do not expose the concurrency in any circumstance—for example, when a program applies the operation to several instances of the same data.

Another widely studied approach is automatic parallelization, where a compiler attempts to find parallelism, typically in programs written in a conventional language such as Fortran. As appealing as it may seem, this approach has not worked well in practice. Accurate program analysis is necessary to understand a program's potential behavior. This analysis is challenging for simple languages such as Fortran, and far more difficult for languages, such as C, that manipulate pointer-based data. Moreover, sequential programs often use sequential algorithms and contain little concurrency.

IMPERATIVE AND FUNCTIONAL LANGUAGES.

Popular commercial programming languages (e.g., Pascal, C, C++, Java, C#) are imperative languages in which a

Software and the Concurrency Revolution



programmer specifies step-by-step changes to variables and data structures. Fine-grained control constructs (e.g., for loops), low-level data manipulations, and shared mutable object instances make programs in these languages difficult to analyze and automatically parallelize.

The common belief is that functional languages, such as Scheme, ML, or Haskell, could eliminate this difficulty because they are naturally suited to concurrency. Programs written in these languages manipulate immutable object instances, which pose no concurrency hazards. Moreover, without side effects, programs seem to have fewer constraints on execution order.

In practice, however, functional languages are not necessarily conducive to concurrency. The parallelism exposed in functional programs is typically at the level of procedure calls, which is impractically fine-grained for conventional parallel processors. Moreover, most functional languages allow some side effects to mutable state, and code that uses these features is difficult to parallelize automatically.

These languages reintroduce mutable state for reasons of expressibility and efficiency. In a purely functional language, aggregate data structures, such as arrays or trees, are updated by producing a copy containing a modified value. This technique is semantically attractive but can be terrible for performance (linear algorithms easily become quadratic). In addition, functional updates do nothing to discourage the writing of a strictly sequential algorithm, in which each operation waits until the previous operation updates the program's state.

The real contribution of functional languages to concurrency comes in the higher-level programming style commonly employed in these languages, in which operations such as map or map-reduce apply computations to all elements of an aggregate data structure. These higher-level operations are rich sources of concurrency. This style of programming, fortunately, is not inherently tied to functional languages, but is valuable in imperative programs.

For example, Google Fellows Jeffrey Dean and Sanjay

Ghemawat describe how Google uses Map-Reduce to express large-scale distributed computations.³ Imperative languages can judiciously add functional style extensions and thereby benefit from those features. This is important because the industry can't just start over. To preserve the huge investment in the world's current software, it is essential to incrementally add support for concurrency, while preserving software developers' expertise and training in imperative languages.

BETTER ABSTRACTIONS

Most of today's languages offer explicit programming at the level of **threads** and **locks**. These abstractions are low-level and difficult to reason about systematically. Because these constructs are a poor basis for building abstractions, they encourage multithreaded programming with its problems of arbitrary blocking and reentrancy.

Higher-level abstractions allow programmers to express tasks with inherent concurrency, which a runtime system can then combine and schedule to fit the hardware on the actual machine. This will enable applications that perform better on newer hardware. In addition, for mainstream development, programmers will value the illusion of sequential execution within a task.

Two basic examples of higher-level abstractions are asynchronous calls and futures. An *asynchronous call* is a function or method call that is nonblocking. The caller continues executing and, conceptually, a message is sent to a task, or fork, to execute operation independently. A *future* is a mechanism for returning a result from an asynchronous call; it is a placeholder for the value that has not yet materialized.

Another example of a higher-level abstraction is an *active object*, which conceptually runs on its own thread so that creating 1,000 such objects conceptually creates 1,000 potential threads of execution. An active object behaves as a monitor, in that only one method of the object executes at a given time, but it requires no traditional locking. Rather, method calls from outside an active object are asynchronous messages, marshaled, queued, and pumped by the object. Active objects have many designs, from specialized actor languages to COM single-threaded apartments callable from traditional C code, and many design variables.

Other higher-level abstractions are needed, such as protocols to describe and check asynchronous message exchange. Together they should bring together a consistent programming model that can express typical application concurrency requirements across all of the major granularity levels.

WHAT WE NEED IN TOOLS

Parallel programming, because of its unfamiliarity and intrinsic difficulty, is going to require better programming tools to systematically find defects, help debug programs, find performance bottlenecks, and aid in testing. Without these tools, concurrency will become an impediment that reduces developer and tester productivity and makes concurrent software more expensive and of lower quality.

Concurrency introduces new types of programming errors, beyond those all too familiar in sequential code. Data races (resulting from inadequate synchronization and deadlocks) and livelocks (resulting from improper synchronization) are difficult defects to find and understand, since their behavior is often nondeterministic and difficult to reproduce. Conventional methods of debugging, such as reexecuting a program with a breakpoint set earlier in its execution, do not work well for concurrent programs whose execution paths and behaviors may vary from one execution to the next.

Systematic defect detection tools are extremely valuable in this world. These tools use static program analysis to systematically explore *all* possible executions of a program; thus, they can catch errors that are impossible to reproduce. Although similar techniques, such as model checking, have been used with great success for finding defects in hardware, which is inherently concurrent, software is more difficult. The state space of a typical program is far larger than that of most hardware, so techniques that systematically explore an artifact's states have much more work to do. In both cases, modularity and abstraction are the keys to making the analysis tractable. In hardware model testing, if you can break off the ALU (arithmetic logic unit) and analyze it independently of the register file, your task becomes much more tractable.

That brings us to a second reason why software is more difficulty to analyze: it is far harder to carve off pieces of a program, analyze them in isolation, and then combine the results to see how they work together. Shared state, unspecified interfaces, and undocumented interactions make this task much more challenging for software.

Defect detection tools for concurrent software comprise an active area of research. One promising technique from Microsoft Research called KISS (Keep it Strictly Sequential)⁴ transforms a threaded program into a sequential program whose execution behavior includes all possible interleaves of the original threads that involve no more than two context switches. The transformed program can then be analyzed by the large number of existing sequential tools, which then become concurrent defect detection tools for this bounded model.

Even with advances such as these, programmers are still going to need good debuggers that let them understand the complex and difficult-to-reproduce interactions in their parallel programs. There are two general techniques for collecting this information. The first is better logging facilities that track which messages were sent to which process or which thread accessed which object, so that a developer can look back and understand a program's partially ordered execution. Developers will also want the ability to follow causality trails across threads (e.g., which messages to one active object, when executed, led to which other messages to other active objects?), replay and reorder messages in queues, step through asynchronous call patterns including callbacks, and otherwise inspect the concurrent execution of their code. The second approach is reverse execution, which permits a programmer to back up in a program's execution history and reexecute some code. Replay debugging is an old idea, but its cost and complexity have been

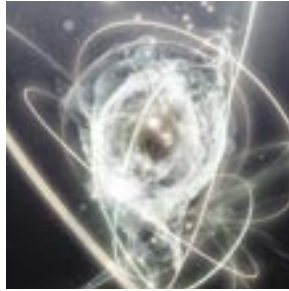


barriers to adoption. Recently, virtual machine monitors have reduced both factors.⁵ In a concurrent world, this technique will likely become a necessity.

Performance debugging and tuning will require new tools in a concurrent world as well. Concurrency introduces new performance bottlenecks, such as lock contention, cache coherence overheads, and lock convoys, which are often difficult to identify with simple profilers. New tools that are more aware of the underlying computer architecture and the concurrent structure of a program will be better able to identify these problems.

Testing, too, must change. Concurrent programs, because of their nondeterministic behaviors, are more difficult to test. Simple code coverage metrics, which track whether a statement or branch has executed, need to be extended to take into account the other code that is executing concurrently, or else testing will provide

Software and the Concurrency Revolution



an unrealistically optimistic picture of how completely a program has been exercised. Moreover, simple stress tests will need to be augmented by more systematic techniques that use model-checking-like techniques to explore systems' state spaces. For example, Verisoft has been very successful in using these techniques to find errors in concurrent telephone switching software.⁶ Today, many concurrent applications use length of stress testing to gain confidence that the application is unlikely to contain serious races. In the future, that will increasingly be insufficient, and software developers will need to be able to prove their product's quality through rigorous deterministic testing instead of relying on a probabilistic confidence based on stress tests.

PARALLELISM IS KEY

The concurrency revolution is primarily a software revolution. The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance.

The software industry needs to get back into the state where existing applications run faster on new hardware. To do that, we must begin writing concurrent applications containing at least dozens, and preferably hundreds, of separable tasks (not all of which need be active at a given point).

Concurrency also opens the possibility of new, richer computer interfaces and far more robust and functional software. This requires a new burst of imagination to find and exploit new uses for the exponentially increasing potential of new processors.

To enable such applications, programming language designers, system builders, and programming tool creators need to start thinking seriously about parallelism and find techniques better than the low-level tools of threads and explicit synchronization that are today's basic building blocks of parallel programs. We need higher-level parallel constructs that more clearly express a programmer's intent, so that the parallel architecture of a

program is more visible, easily understood, and verifiable by tools. Q

REFERENCES

1. Sutter, H. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal* 30 (3); <http://www.gotw.ca/publications/concurrency-ddj.htm>.
2. Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems* 22 (2): 416-430.
3. Dean, J., and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, San Francisco, CA: 137-150.
4. Qadeer, S., and Wu, D. 2004. KISS: Keep it Simple and Sequential. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington, DC: 1-13.
5. King, S. T., Dunlap, G. W., and Chen, P. M. 2005. Debugging operating systems with time-traveling virtual machines. *Proceedings of the 2005 Annual Usenix Technical Conference*, Anaheim, CA: 1-15.
6. Chandra, S., Godefroid, P., and Palm, C. 2002. Software model checking in practice: an industrial case study. *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL: 431-441.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

HERB SUTTER is a software architect in Microsoft's developer division. He chairs the ISO C++ standards committee, and is the author of four books and more than 200 technical papers and articles, including the widely read "The Free Lunch Is Over" essay on the concurrency revolution. He can be reached at hsutter@microsoft.com.

JAMES LARUS is a senior researcher at Microsoft Research, managing SWIG (Software Improvement Group), which consists of the SPT (software productivity tools), TVM (testing, verification, and measurement), and HIP (human interactions in programming) research groups, and running the Singularity research project. Before joining Microsoft, he was an associate professor at the University of Wisconsin-Madison, where he co-led the Wisconsin Wind Tunnel research project. This DARPA- and NSF-funded project investigated new approaches to building and programming parallel shared-memory computers. Larus received his Ph.D. in computer science from the University of California at Berkeley.

© 2005 ACM 1542-7730/05/0900 \$5.00

Continued from page 64

multicore CPU-based desktop systems will stall as customers figure out that most of their applications run no faster on a dual- or quad-core system than on a uniprocessor system. To sell more machines/CPUs, hardware vendors will have to do what Sun did and “encourage” application vendors to redesign their applications to be MT-hot. Desktop application vendors who have been able to depend on continual CPU clock increases will now have to invest in a long and painful rewrite of their software to gain the next jump in performance and functionality. All this could take years. Moreover, more agile companies will now have an opening to make MT-hot investments faster, potentially snagging customers from incumbent vendors that are too slow to make the transition.

What is frustrating is that all of this could have been avoided. MT has been on the horizon for at least a decade. Because technology companies take a myopic quarter-by-quarter view in their planning, they missed the bigger trend of multicore CPUs and their implications for the desktop. As a result, the tools for MT development are not in place as these new CPUs hit the market. With the exception of Java’s minimal MT support, things look fairly close to what the large enterprise application developers had to work with more than 10 years ago.

Sadly, I see the following scenario playing out. It will take several years of pain for application developers to rewrite their code to be MT-hot. Once a methodology for conversion has been established, IDE tool vendors will start bringing out automation extensions that help manage MT development complexity. These two processes could easily take three to five years. Once MT-enhanced IDE products become established, language extensions will follow. A commercially accepted development language with fully integrated MT control structures should come into widespread use in five to seven years. In the meantime, don’t count on that instant performance increase for desktop applications with the release of each new CPU family. With multicore systems, having CPU bandwidth on the desktop and being able to use it are going to be two very different things. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MACHE CREEGER (mache@creeger.com) is a 30-year veteran of the technology industry. He is the principal of Emergent Technology Associates, marketing and business development consultants to technology companies.

© 2005 ACM 1542-7730/05/0900 \$5.00



Career Resource Centre



The smart way to manage your career

Looking for your next IT job? Need career advice?

Visit the new ACM Career Resource Centre!

ACM Members enjoy full access to the CRC!

- View and apply for Jobs ♦ Perform advanced searches ♦ Advertise your resume for employers to view
- ♦ Maintain your personal account ♦ Receive email notification as matches are posted ♦ Manage your career planning ♦ Refine your professional skills ♦ Master job search techniques ♦ Discover long-range IT trends
- ♦ Explore first-hand accounts from pros in the field ♦ Participate in career and technical forums ♦ Use exclusive self-assessment tools to pinpoint strengths and weaknesses, develop career paths, and more!

New —

CareerNews, ACM’s biweekly digest of career news and resources for professionals and students!

For details on how ACM can help you manage your IT career, go to www.acm.org/crc



Association for Computing Machinery
The First Society in Computing
www.acm.org

AD25



Multicore CPUs for the Masses

Mache Creeger, Emergent Technology Associates

Multicore is the new hot topic in the latest round of CPUs from Intel, AMD, Sun, etc. With clock speed increases becoming more and more difficult to achieve, vendors have turned to multicore CPUs as the best way to gain additional performance. Customers are excited about the promise of more performance through parallel processors for the same real estate investment.

For a handful of popular server-based enterprise applications, that may be true, but for desktop applications I wouldn't depend on that promise being fulfilled anytime soon. The expectation for multicore CPUs on the desktop is to have all our desktop applications fully using all the processor cores on the chip. Each application would gracefully increase its performance as more and more processors became available for use. Just like past increases in clock speed and application bandwidth, increasing the number of processor cores should produce similar performance enhancements. It works for the popular enterprise applications, so why not for desktop applications? Sounds reasonable, right? Don't count on it.

Sure, the major enterprise applications such as Oracle, WebLogic, DB2, and Apache are designed to take full advantage of multiple processors and are architected to be MT (multithreaded). They have to be for the large SMP (symmetric multiprocessing) servers that are the meat and potatoes of their market.

Even though the concept of using concurrent CPUs to increase overall software performance has been around for at least 35 years, remarkably little in the way of development tools has made it to the commercial marketplace. As a result, the vast majority of applications are single-threaded. Although multicore CPUs will allow you to share a mix of applications across multiple processors, individual application performance will remain bounded by the speed of an individual processor. Application performance will remain the same regardless of whether you have one or 100 processors because each application can run on only one processor at any given time.

With the possible exception of Java, there are no widely used commercial development languages with MT extensions. Realistically, until now there has not been much of a need. The widespread availability of com-

Will increased

CPU BANDWIDTH

TRANSLATE INTO USABLE

DESKTOP PERFORMANCE?

mercial SMP systems did not really arrive until the early 1990s, and even then multithreaded applications came slowly.

When I was at Sun, the company rewrote SunOS to take advantage of its new multithreading architecture. It was a long and painful process. Initially, subsystems were rewritten with locks at either end so they would be assured to run as one big single thread (MT-safe) and then rewritten again to be fully MT optimized (MT-hot) for maximal concurrency. Everything was designed by hand and there were no tools to manage the complexity.

Around the same time, Sun implemented a set of user MT libraries that applications could use. As larger SMP servers started to appear on Sun's roadmap, the major enterprise application vendors saw that they too had to make the investment in converting their software to MT. The experience was equally painful and similar to the SunOS MT rewrite. Recognizing the need to make these applications run MT-hot in order to sell their new SMP servers, Sun leveraged its experience by assigning engineers to these companies to help them in their migration.

The situation today is quickly becoming a replay of what happened 10 years ago. Application vendors requiring more CPU bandwidth can no longer count on increased clock speeds for better performance and functionality. Most large-scale client-side applications are written in C or C++ and historically have been designed to be single-threaded. Making applications MT-hot is still a labor-intensive redelivery process. Although a few vendors, most notably in the multimedia area, have made some MT enhancements to their applications, they have just started to pick off the low-hanging fruit. With multicore CPUs, widespread desktop performance and functionality improvements are still years away.

What have the development tool vendors been doing as MT architectures have evolved during the past decade or so? It's not as if anyone in the computer industry did not see this coming. What can we expect in the future? Given where the industry is today, the introduction of

Continued on page 63

Improve Your Process, Start Testing Now

Once you start using our tools, you are well on your way to building rock solid applications, in the shortest possible time, **guaranteed!**

Get your free trial at www.automatedqa.com/downloads

Award Winning Testing Solutions

TestComplete

"AutomatedQA's TestComplete is a great product and just as capable as the market leader at less than one-tenth the price. Why would anybody pay \$6000 per seat for test automation?"

Joel Spolsky

- GUI Testing
- Unit Testing
- Regression Testing
- Functional Testing
- Load Testing
- Distributed Testing
- Web Testing
- Data-Driven Testing
- Object-Driven Testing

Performance and Memory Profiling

AQtime

- Managed/Unmanaged Applications
- Performance Analysis
- Test Coverage
- Memory Analysis
- Line-level Precision
- Microsoft Visual Studio .NET Integration

Issue Tracking and Project Management

AQdevTeam

- Configurable Workflows
- Configurable Fields
- Configurable Forms
- Scriptable Macros
- Email Notifications
- Web Interface
- Windows Interface

Build Automation and Release Management

Automated Build Studio

- Visual Macro Builder
- Extensible Actions
- Scheduled Builds
- Test Integration
- Issue Tracking Integration
- Microsoft Visual Studio .NET Integration



AutomatedQA
www.automatedqa.com

(702) 891-9424

All AutomatedQA products include a **60 day money-back guarantee**



YOU VS THE INCREDIBLE SHRINKING DEADLINE



Rational

The pressure's building. How do you spur innovation? Get IBM Rational, a powerful set of integrated development tools supporting asset-based development. It's part of **AN IBM MIDDLEWARE SOLUTION THAT'LL HELP YOU CREATE BETTER SOFTWARE FASTER**. Based on the Eclipse™ open-source platform, Rational supports and runs across multiple platforms, including Linux®. So you and your teams work faster and smarter. And that always takes the top title.

SEE FOR YOURSELF: VISIT IBM.COM/MIDDLEWARE/TOOLS TO DOWNLOAD TRIAL VERSIONS OF RATIONAL SOFTWARE MODELER AND RATIONAL SOFTWARE ARCHITECT

IBM, the IBM logo and Rational are registered trademarks or trademarks of International Business Machines Corporation in the United States and/or other countries. Eclipse is a trademark of Eclipse Foundation, Inc. Linux is a registered trademark of Linus Torvalds. ©2005 IBM Corporation. All rights reserved.