
We Trip The Light
FANTASTIC

CMP
United Business Media

Java • TiVo • Python • C++ • Netbeans • .NET • DSP

#370 MARCH 2005

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

http://www.ddj.com

64-BIT COMPUTING

64-Bit Computing & JVM Performance

Windows & The World of 64-Bit Computing

Integer 64-Bit Optimizations

64-Bit Computing & DSPs

Moving Toward Concurrency in Software

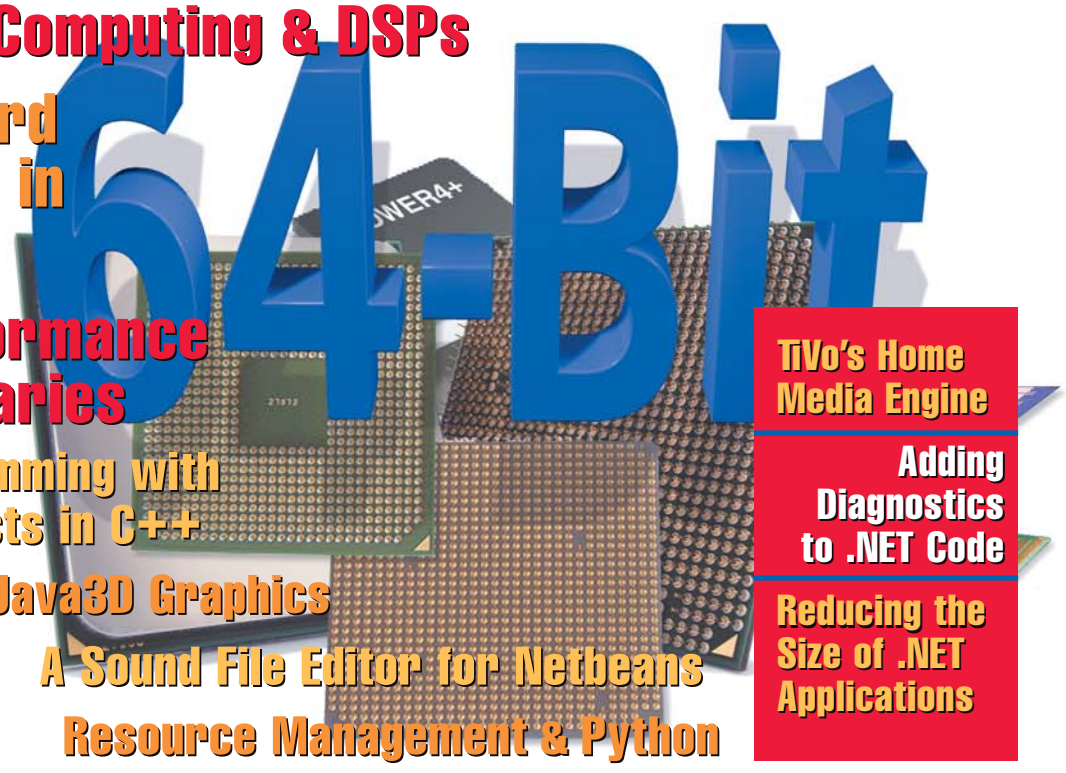
High-Performance Math Libraries

Programming with Contracts in C++

Java3D Graphics

A Sound File Editor for Netbeans

Resource Management & Python



TiVo's Home Media Engine

Adding Diagnostics to .NET Code

Reducing the Size of .NET Applications

CONTENTS

MARCH 2005 VOLUME 30, ISSUE 3

FEATURES

EMBEDDED SYSTEMS

COLUMNS

FORUM

RESOURCE CENTER

As a service to our readers, source code, related files, and author guidelines are available at http://www.ddj.com/. Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@ddj.com, faxed to 650-513-4618, or mailed to *Dr. Dobb's Journal*, 2800 Campus Drive, San Mateo CA 94403.

For subscription questions, call 800-456-1215 (U.S. or Canada). For all other countries, call 902-563-4753 or fax 902-563-4807. E-mail subscription questions to ddj@neodata .com or write to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. If you want to change the information you receive from CMP and others about products and services, go to http://www.cmp.com/feedback/permission.html or contact Customer Service at the address/number noted on this page.

Back issues may be purchased for $9.00 per copy (which includes shipping and handling). For issue availability, send e-mail to orders@cmp.com, fax to 785-838-7566, or call 800-444-4881 (U.S. and Canada) or 785-838-7500 (all other countries). Back issue orders must be prepaid. Please send payment to *Dr. Dobb's Journal*, 4601 West 6th Street, Suite B, Lawrence, KS 66049-4189. Individual back articles may be purchased electronically at http://www.ddj.com/.

NEXT MONTH: We'll be focusing on Internet and web development in the April issue.

DR. DOBB'S JOURNAL (ISSN 1044-789X) is published monthly by CMP Media LLC., 600 Harrison Street, San Francisco, CA 94017; 415-947-6000. Periodicals Postage Paid at San Francisco and at additional mailing offices. SUBSCRIPTION: $34.95 for 1 year; $69.90 for 2 years. International orders must be prepaid. Payment may be made via Mastercard, Visa, or American Express; or via U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year. U.K. subscribers contact Jill Sutcliffe at Parkway Gordon 01-49-1875-386. POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80328-6188. Registered for GST as CMP Media LLC, GST #13288078, Customer #2116057, Agreement #40011901. INTERNATIONAL NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 30 Montgomery St., Jersey City, NJ 07302; 212-332-7100.

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

DR. DOBB'S JOURNAL
2800 Campus Drive, San Mateo, CA 94403
650-513-4300. http://www.ddj.com/

CMP MEDIA LLC
Gary Marshall President and CEO
John Day Executive Vice President and CFO
Steve Weitzner Executive Vice President and COO
Jeff Patterson Executive Vice President, Corporate Sales & Marketing
Mike Mikos Chief Information Officer
William Amstutz Senior Vice President, Operations
Leah Landro Senior Vice President, Human Resources
Mike Azzara Vice President/Group Director Internet Business
Sandra Grayson Vice President & General Counsel
Alexandra Raine Vice President Communications
Robert Faletra President, Channel Group
Vicki Masseria President CMP Healthcare Media
Philip Chapnick Vice President, Group Publisher Applied Technologies
Michael Friedenberg Vice President, Group Publisher InformationWeek Media Network
Paul Miller Vice President, Group Publisher Electronics
Fritz Nelson Vice President, Group Publisher Network Computing Enterprise Architecture Group
Peter Westerman Vice President, Group Publisher Software Development Media
Joeseph Braue Vice President, Director of Custom Integrated Media Solutions
Shannon Aronson Corporate Director, Audience Development
Michael Zane Corporate Director, Audience Development
Marie Myers Corporate Director, Publishing Services

CMP
United Business Media
ABP American Buisness Press
Printed in the USA
BPA

Smart Stuff

When it comes to marketing, the smart thing to do seems to be to prefix "smart" to whatever is being pitched. Let's see, there are smart cards, smart phones, smart cars, smart growth, smart dust, smart architectures, and smart yada yada yada. In our neck of the woods, you can get smart compilers, smart debuggers, and smart linkers. The only thing you can't get, at least according to what my boss recently told me, is smart editors. But ha, ha—the joke was on him, as I quickly pointed to Smart Editor Professional 3.0 at http://tucows.tr.net/preview/362587.html. Then there's Gene Smarte, my old boss at *BYTE* magazine, and of course, Maxwell Smart, who moved from TV reruns to CIA archives (http://www.cia.gov/spy_fi/item15.html).

Smart houses seem to be the coming thing. A lab/house created by Eneo Labs (http://www.eneo.com/eng/), for instance, can clean itself via baseboard automatic vacuum cleaners, adjust to weather changes thanks to a roof-top weather station, and cut energy consumption as needed. And, as you might expect, security and entertainment are central to the home. Electronic keys let you open doors and security cameras help you keep an eye on the kids. Large-screen TV displays throughout the house allow you to watch TV or interact with the central server, which stores movies, TV shows, MP3 files, and the like.

At the heart of the "Connected House" is Eneo's IPbox, an embedded computer that serves as the residential gateway with broadband access and eight Ethernet ports, Wi-Fi access, audiovisual interface, and universal remote control. The OSGi-based network software is called "eNeo NET" and, among other things, it takes care of incompatibilities between devices. The browser-based interface makes it possible for inhabitants to access and manage services from a TV set, PDA, cell phone, or PC.

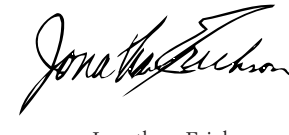
But if projects like Eneo's Connected House were nothing more than toy-houses for the rich and famous, they wouldn't be worth wasting the space. However, smart houses do have practical purposes, especially when enabling assistive care for disabled and/or elderly inhabitants. Domotic systems that use PDAs, cell phones, sensors, and Internet access are being used for everything from alerting emergency services to unlocking the front door, making it possible for all of us to live fuller lives. In this context, a smart house is a smart idea.

To my mind, another smart idea is that of smart guns. Of course, there are few public health issues that are more controversial than the hint of firearm regulation. Shoot, my e-mail box is already overflowing from just that sentence alone. But every year, according to the Centers for Disease Control, more than 30,000 people in the U.S. die from firearms-related deaths. Of that number, about two people a day are killed by accidental gun discharges. Smart guns, which involve an electronic means of authenticating the user, are one approach to addressing this problem.

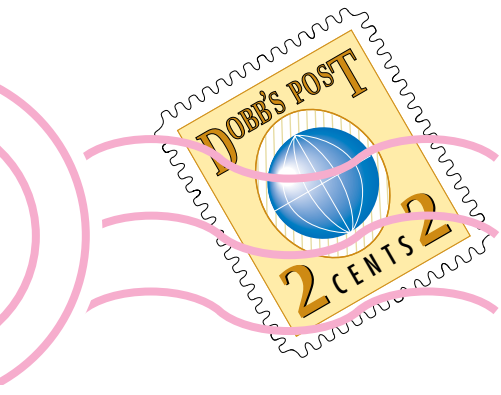
The latest in smart-gun technology was recently demonstrated at the New Jersey Institute of Technology (http://www.njit.edu/). In this case, "dynamic grip recognition," a technology invented by NJIT professor Michael Recce, was implemented by NJIT professor Timothy Chang who embedded 20 small electronic sensors in the handle of a gun. The gun is then "trained" as to who is authorized to use the firearm—and there can be dozens of user profiles stored in the gun—by "learning" the physical markers and behavior of authorized users. The biometric technique measures not only the size, strength, and structure of a person's hand, but also the reflexive way in which the person acts. For smart guns, the observed actions are how the person squeezes something, such as a trigger, to produce a unique and measurable pattern. Sensors in the experimental gun then can read and record the size and force of the users' hand during the first second when the trigger is squeezed. The sensors currently being used are off-the-shelf 4.5mm-diameter discrete piezoelectric ceramic-disk sensors. NJIT is working with a sensor company to put custom conformal capacitive sensor arrays into the grips. Profile patterns that execute recognition algorithms are stored in SRAM. For the time being, a standard serial port interfaces to the PC because the DSP system is a standalone unit. However, the next version will be untethered and use a Bluetooth interface.

It's worth noting that Recce's dynamic grip-recognition technology is not limited just to guns, but might also be applied to, say, a car's steering wheel to prevent theft or other misuse. NJIT will now turn over the prototype to Metal Storm (http://www.metalstorm.com/), which will incorporate NJIT technology into its patented electronic handgun.

Of course, you can go too far with the "smart" moniker. There's smart government, smart drugs, smart bombs, McDonald's smart meals, smart parents (according to teenagers), and smart beer. For me, the smart thing to do now is sign off.



Jonathan Erickson
editor-in-chief
jerickson@ddj.com



Licensing & Such

Dear *DDJ*,

The same day I read Jan Galkowski's letter in the January 2005 issue of *DDJ*, I watched a TV news story that included the following information:

When the Insurance Institute for Traffic Safety crash-tested the 2005 Hyundai Elantra, the driver's air bag failed to deploy. According to their press release, "Hyundai engineers will modify the software that determines whether and when to fire the airbags in 2005 models built after December 2004. The company also will recall cars manufactured earlier to fix this problem." (To refresh my memory, I looked this up at http://www.hwysafety.org/news_releases/2004/pr121904.htm.)

Why do you suppose the Hyundai "engineers" didn't get the software right the first time? I suspect they wrote it the same way most software is written, waiting for (to use Jan's words) "end-user critiques, complaints, and bug reports" to reveal "incorrect expectations or documentation."

I agree with Jan that this typical software-development process is not desirable, but I don't think we necessarily have to tolerate it. I think we can make the case that there are times when software-development standards should take precedence over our employers' need to be successful. I also believe that adopting standards will, ironically, make many businesses more successful in the long run.

Software that is intended to help people make decisions usually comes with a disclaimer that the vendor is not liable for the users' bad decisions. However, software that actually makes decisions, such as when to deploy an airbag, should be held to a higher standard. The owner of a construction company would expect a building engineer to refuse to proceed if appropriate specifications weren't available. Otherwise, they both might get sued for violating well-established standards, i.e., building codes. I agree with Brent Fulgham (*DDJ*, August 2004. "Letters") that the public will eventually demand standards for the development of software that affects public safety. I don't think these standards should be required for other software, but they should be strongly recommended.

I agree that we would have to grandfather existing software. Automakers, for example, would not be required to recall every car that uses old software, just those that turned out to be defective using current testing methods. However, I don't think it is ridiculous to expect them to start using standards to write or rewrite any software used in new cars. As for outsourcing, it wouldn't matter where the software was written. If the car were driven in the U.S. (or any country that adopted standards), the standards would apply.

There is a precedent for holding companies accountable for their software-development process. In my June 2004 *DDJ* letter to the editor, I noted that the FDA inspects the software specification and documentation procedures of medical equipment manufacturers. (See http://www.eweek.com/article2/0,1759,1543652,00.asp?kc=EWNWS030804DTX1K0000599 for more information.) However, as Jan points out, there's not much hope of finding acceptable procedures as long as the people who are supposed to know what the software is expected to do cannot describe these expectations in sufficient detail.

I have colleagues who believe that users are so stupid that they deserve the lousy software they get. I disagree. It has been my experience that business people can be taught how to create sufficiently detailed specifications. Accurate and complete specifications not only lead to better software, but often lead to business improvements that have nothing to do with software. However, accurate and complete specifications are time consuming to create and have little short-term benefit. Since decision makers are usually unwilling to wait for long-term benefits, I don't expect most of them to adopt this approach unless they are forced to (which they should be if they produce products that affect public safety). Once standards have been established, however, some businesses that have nothing to do with public safety will realize the long-term financial benefits of standards-based software development. Unfortunately, many will continue to be short-sighted; and I will continue to point this out whenever I encounter them. My long-term goal, by the way, is that someday I will be able to say, "my code is up to code."

Jim Wiggins
jwiggins@ifbf.org

Dynamic Caching

Dear *DDJ*,

I enjoyed the article "Dynamic Caching & ADO DataSet" by John Cheng and Hong Rong (*DDJ*, December 2004), which was a useful introduction to smart caching of larger datasets and pointed out some of the drawbacks of the relatively simple default behavior of ADO.

However, I would like to pick up on an error in the example in the text for an incremental query. If a partial set of employee data ordered by *fname* and *lname* is retrieved and the last record is "Joe Smith," then it is incorrect to request incremental data by adding the clause *where fname > 'Joe' and lname > 'Smith'* to the original query.

Alas, a few seconds thought should demonstrate that neither the original query nor the subsequent one would locate my record, given my first name is "Roger" and my last name is "Orr." The incremental query needs to be more carefully thought through if more than one column is used for the ordering: one solution would be: *where (fname = 'Joe' and lname > 'Smith') or fname > 'Joe')*.

Roger Orr
rogero@howzatt.demon.co.uk

Strange Bedfellows

Dear *DDJ*,

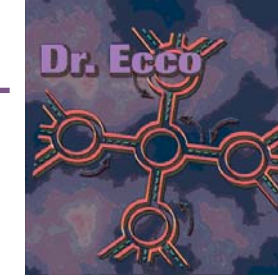
I enjoy *Dr. Dobb's Journal* a great deal. In particular, I read Jonathan Erickson's "Strange Bedfellows" editorial (*DDJ*, December 2004) and I don't get what he doesn't get.

IBM and Open Source are not an unlikely pair— they are *the* most likely pair. IBM, according to publicly disclosed financial results, is now about 75 percent services and hardware, making it the largest hardware and the largest services company in the world rolled into one. Based on this profile, IBM is the most likely company to ally itself with open-source software. Joel Spolsky expounded on this more than two years ago (http://www.joelonsoftware.com/articles/StrategyLetterV.html) and what he wrote then is still basically true now: IBM is commoditizing the complement to its core business.

Imagine a hypothetical future world, 15 years from now, where through twists of fate, there is no software except open-source software. Which of IBM's competitors of 2004 remain in that hypothetical world? Come on, IBM has every reason to embrace and endorse open-source software today. That this fact still surprises people surprises me.

Dino Chiesa
dinoch@microsoft.com

DDJ



Grab Bag

Dennis E. Shasha

Since I have a key to Dr. Ecco's apartment, I walked in one day but found it empty. Ecco had been asked by Baskerhound to solve a seemingly difficult two-person game as part of a code-breaking investigation. Ecco had taken his niece and nephew to a casino that was somehow involved. I was therefore left with this letter from Liane.

Dear Professor Scarlet,

Grab Bag is a simple game to play but difficult to win. The first player is given an empty "seed" collection *Aseed* and a "grab bag" *Agrab*. The second player is given an empty seed collection *Bseed* and a grab bag *Bgrab*. The players agree on a positive number n. Here is the general idea: The players alternate moves, where a move consists of inserting a whole number to one's own seed collection (the same number can be inserted several times). When a player does so, he or she puts into his/her grab bag any number k between 1 and n resulting from adding the just inserted number to an element of the opponent's seed collection, provided k hasn't been previously "grabbed" (that is, put into a grab bag) by either player. When all the numbers between 1 and n are grabbed, the game is over and the player with the most numbers in his/her grab bag wins.

The first move consists of inserting a number between 0 and n. Subsequently, every move consists of inserting a nonnegative number less than or equal to n to the player's seed bag, and results in putting at least one ungrabbed number between 1 and n in his/her grab bag, if it is possible for the player to do so. If not possible, but there are ungrabbed numbers left, then the player may use a seed number between $-n$ and -1 to grab a number. You can prove that it is always possible to grab a number if there are any ungrabbed ones left; the player is obligated to grab on every move.

Warm-up: Who wins when $n=3$?

Solution Idea:

a. In A's first move, A cannot grab anything but must choose one of 1, 2, or 3 as a seed.
b. B can respond by grabbing one.
c. A can then grab at most one other number, because B has only one seed.
d. B can then grab the last.

But life is not always so straightforward. For example, A can force a draw when $n=4$, but only if he/she prevents B from grabbing two numbers in B's second move.

Now here are the questions, Professor:

1. Can either player force a win when $n=5$?
2. Is there a winning strategy for either player, depending on the value of n?
3. Do any of these answers change if the players could use negative seed numbers on any turn, including when it is possible to grab a number with a nonnegative seed? For this scenario, we'll add a new rule: If a player blocks the game (that is, makes a move that prevents the opponent from grabbing a number when there are ungrabbed numbers left), then the blocking player loses."

For the solution to last month's puzzle, see page 86.

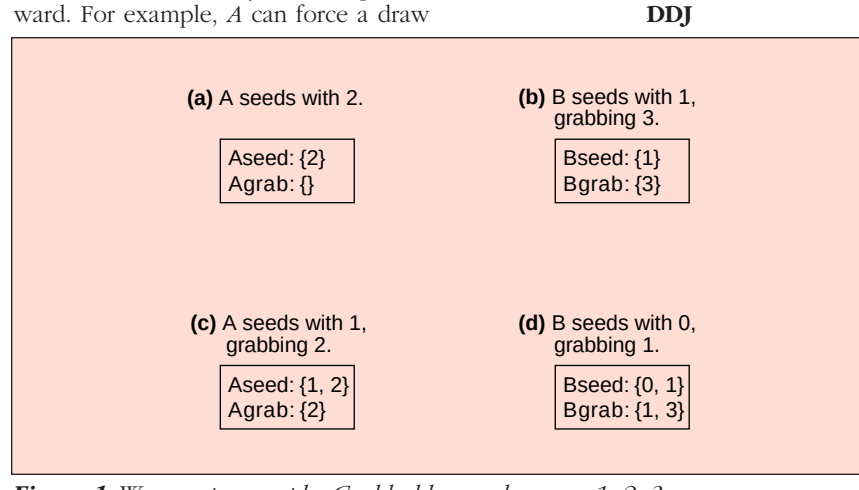


Figure 1: *Warm-up example: Grabbable numbers are 1, 2, 3.*

Dennis is a professor of computer science at the Courant Institute, New York University. His latest books include Dr. Ecco's Cyberpuzzles: 36 Puzzles for Hackers and Other Mathematical Detectives *(W.W. Norton, 2002) and* Database Tuning: Principles, Experiments, and Troubleshooting Techniques *(Morgan Kaufman, 2002). He can be contacted at DrEcco@ddj.com.*

Assistive Technologies

Researchers at the University of California, Santa Cruz (http://www.ucsc.edu/), are developing new assistive technologies for the blind based on advances in computer vision that have emerged from research in robotics. A "virtual white cane" is one of several prototype tools for the visually impaired developed by Roberto Manduchi, an assistant professor of computer engineering, and his students.

Manduchi's alternative to the traditional white cane is a laser-based range-sensing device about the size of a flashlight. A laser is combined with a digital camera and CPU that analyzes and integrates spatial information as users move the device back and forth over a scene. Users receive feedback about the scene in the form of audio signals, and an additional tactile interface is being developed for future prototypes.

Dan Yuan, a graduate student working with Manduchi on the virtual white cane project, built the initial prototype. The UCSC researchers are collaborating with the Smith-Kettlewell Eye Research Institute, a nonprofit research institute in San Francisco (http://www.ski.org/), on the virtual white cane and other projects, such as a project Manduchi refers to as "MapQuest for the blind." The project hopes to create a feedback environment so that blind people can explore maps on the computer. The feedback would be provided by a "force-feedback mouse," which vibrates to produce a variety of physical sensations users can feel as the pointer moves across features on a computer screen. These devices are readily available, so the project involves creating software that will enable the blind to use a force-feedback mouse to "feel" their way through a map.

Python 2.4 Released

The Python Software Foundation has announced the release of Python 2.4 (from http://www.python.org/2.4/). New language features include multiline imports; failed import cleanup; function/method decorators; and an *-m* command-line option, which invokes a module in the standard library. Additionally, Python no longer generates Overflow warnings, and the compiler now treats assigning to None as a SyntaxError.

New language features, however, are not the focus of the release. Instead, Python 2.4 concentrates on performance enhancements and ease-of-use improvements. Several optimizations have been added to the interpreter, and some modules new in Python 2.3—including sets and heapq—have been recoded in C. Python 2.4 runs the pystone benchmark 5 percent faster than Version 2.3, and 35 percent faster than Python 2.2.

The Web According to TAG

The W3C's Technical Architecture Group (TAG) has completed an ambitious Recommendation titled "Architecture of the World Wide Web, Volume One" (published at http://www.w3.org/TR/2004/REC-webarch-20041215/#app-principles). The document sets out "core-design components, constraints, and good practices…by software developers, content authors, site managers, users, and specification designers."

Tim Berners-Lee is cochair of the TAG, along with HP's Stuart Williams; other participants in the group are drawn from Microsoft, IBM, Sun, Day Software, and the W3C.

Those Blooming Cell Phones

Motorola, the University of Warwick, and the manufacturing company PVAXX Research & Development say they have jointly developed a mobile phone cover that sprouts into a sunflower if it ends up in a landfill. A special biodegradable polymer that looks like ordinary plastic was used for the case, and it was embedded with a small transparent window containing a dwarf sunflower (http://www2.warwick.ac.uk/newsandevents/pressreleases/NE1000000097300/).

Motorola has not officially committed to bringing the cell phone cover into production, but said that products using the new polyvinylalcohol polymer could be on the market as early as midsummer. PVAXX says the biodegradable material can be made rigid or flexible in shape.

While the U.S. Environmental Protection Agency estimates that discarded electronic equipment now comprises only 1 or 2 percent of the 210 million tons of solid waste the U.S. produces annually, that number is expected to rise dramatically over the next few years. The European Union requires mobile handset manufacturers to eliminate toxic substances (mercury, lead, and brominated flame retardants) from their mobile handsets by 2006, and has set a cell phone recycling/reuse target of 65 percent.

Neural-Based Sensor System Identifies Gun Shots

The Smart Sensor Enabled Threat Recognition and Identification (Sentri) system combines video cameras, microphones, computers, and software modeled after neural sound processing to identify gunshots, pinpoint their location, and relay the coordinates to a command center. Developed by Theodore Berger at the University of Southern California's Center for Neural Engineering (http://www.usc.edu/dept/engineering/CNE/), the software uses wavelet analysis to divide sound into fragments, then match fragments to established audio-wave patterns, while still analyzing the incoming noise. Sentri uses acoustic recognizers, posted in groupings on utility poles, which are tuned to certain specific warning sounds with extremely high accuracy. A directional analyzer calculates any authenticated gunshot's location, using the difference in the time the sound arrives at the different microphones on a Sentri acoustic unit. Field tests with real weapons have shown 95 percent accuracy with respect to gunshot recognition. Chicago is installing the first five of a planned 80 devices in high-crime neighborhoods.

Firefox Browser Blazes On

Internet Explorer's total market share has dipped below 90 percent for the first time in years, according to one web analytics firm, while the open-source browser Firefox is accelerating in popularity. OneStat.com reports that Mozilla's browsers now have a total global usage share of 7.35 percent—up from 2.1 percent at the end of May—while Internet Explorer has slipped five points to 88.9 percent.

What's more, the two browsers developed by the Mozilla Foundation, Mozilla and Firefox, don't appear to be competing with each other. OneStat.com noted a small uptick in the number of Mozilla users at the same time that the new Firefox user base appeared. Instead, it appears that most new Firefox users are previous Internet Explorer users.

While the exact numbers are disputed—OneStat.com's rival WebSideStory pegs Internet Explorer's market share at 91.8 percent—analysts agree that Firefox's momentum is continuing. According to WebSideStory, Firefox's usage share grew by 13 percent in October and 34 percent in November.

A Fundamental Turn Toward Concurrency in Software

Your free lunch will soon be over. What can you do about it?

HERB SUTTER

Your free lunch will soon be over. What can you do about it? What are you doing about it. The major processor manufacturers and architectures, from Intel and AMD to Sparc and PowerPC, have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds and straight-line instruction throughput ever higher, they are instead turning en masse to hyperthreading and multicore architectures. Both of these features are available on chips today; in particular, multicore is available on current PowerPC and Sparc IV processors, and is coming in 2005 from Intel and AMD. Indeed, the big theme of the 2004 In-Stat/MDR Fall Processor Forum (http://www.mdronline.com/fpf04/index.html) was multicore devices, with many companies showing new or updated multicore processors. Looking back, it's not much of a stretch to call 2004 the year of multicore.

And that puts us at a fundamental turning point in software development, at least for the next few years and for applications targeting general-purpose desktop computers and low-end servers (which happens to account for the vast bulk of the dollar value of software sold today). In this article, I describe the changing face of hardware, why it suddenly does matter to software, and how it specifically matters to you and changes the way you'll likely be writing software in the future.

Arguably, the free lunch has already been over for a year or two, only we're just now noticing.

The Free Performance Lunch

There's an interesting phenomenon known as "Andy giveth, and Bill taketh away." No matter how fast processors get, software consistently finds new ways to eat up the extra speed. Make a CPU 10 times as fast, and software usually finds 10 times as much to do (or in some cases, will feel at liberty to do it 10 times less efficiently). Most classes of applications have enjoyed free and regular performance gains for several decades, even without releasing new versions or doing anything special because the CPU manufacturers (primarily) and memory and disk manufacturers (secondarily) have reliably enabled ever-newer and ever-faster mainstream systems. Clock speed isn't the only measure of performance, or even necessarily a good one, but it's an instructive one: We're used to seeing 500MHz CPUs give way to 1GHz CPUs, which give way to 2GHz CPUs, and so on. Today, we're in the 3GHz range on mainstream computers.

> "Concurrency is the next major revolution in how we write software"

The key question is: When will it end? After all, Moore's Law predicts exponential growth, and clearly exponential growth can't continue forever before we reach hard physical limits; light isn't getting any faster. The growth must eventually slow down and even end. (Caveat: Yes, Moore's Law applies principally to transistor densities, but the same kind of exponential growth has occurred in related areas such as clock speeds. There's even faster growth in other spaces, most notably the data storage explosion, but that important trend belongs in a different article.)

If you're a software developer, chances are you have already been riding the "free lunch" wave of desktop computer performance. Is your application's performance borderline for some local operations? "Not to worry," the conventional (if suspicious) wisdom goes, "tomorrow's processors will have even more throughput, and anyway, today's applications are increasingly throttled by factors other than CPU throughput and memory speed (for instance, they're often I/O-bound, network-bound, or database-bound)." Right?

Right enough, in the past. But dead wrong for the foreseeable future.

The good news is that processors are going to continue to become more powerful. The bad news is that, at least in the short term, the growth will come mostly in directions that do not take most current applications along for their customary free ride.

Herb Sutter (http://www.gotw.ca/) chairs the ISO C++ Standards committee and is an architect in Microsoft's Developer Division. His most recent books are Exceptional C++ Style *and* C++ Coding Standards *(Addison-Wesley).*

Over the past 30 years, CPU designers have achieved performance gains in three main areas, the first two of which focus on straight-line execution flow:

- Clock speed.
- Execution optimization.
- Cache.

Increasing clock speed is about getting more cycles. Running the CPU faster more or less directly means doing the same work faster.

Optimizing execution flow is about doing more work per cycle. Today's CPUs sport some more powerful instructions, and they perform optimizations that range from the pedestrian to the exotic, including pipelining, branch prediction, executing multiple instructions in the same clock cycle(s), and even reordering the instruction stream for out-of-order execution. These techniques are all designed to make the instructions flow better and/or execute faster, and to squeeze the most work out of each clock cycle by reducing latency and maximizing the work accomplished per clock cycle.

Note that some of what I just called "optimizations" are actually far more than optimizations, in that they can change the meanings of programs and cause visible effects that can break reasonable programmer expectations. This is significant. CPU designers are generally sane and well-adjusted folks who normally wouldn't hurt a fly and wouldn't think of hurting your code…normally. But in recent years, they have been willing to pursue aggressive optimizations just to wring yet more speed out of each cycle, even knowing full well that these aggressive rearrangements could endanger the semantics of your code. Is this Mr. Hyde making an appearance? Not at all. That willingness is simply a clear indicator of the extreme pressure the chip designers face to deliver ever-faster CPUs; they're under so much pressure that they'll risk changing the meaning of your program, and possibly break it, to make it run faster. Two noteworthy examples in this respect are *write* reordering and *read* reordering: Allowing a processor to reorder *write* operations has consequences that are so surprising, and break so many programmer expectations, that the feature generally has to be turned off because it's too difficult for programmers to reason correctly about the meaning of their programs in the presence of arbitrary *write* reordering. Reordering *read* operations can also yield surprising visible effects, but that is more commonly left enabled anyway because it isn't quite as hard on programmers (and the demands for performance cause designers of operating systems and operating environments to compromise and choose models that place a greater burden on programmers because that is viewed as a lesser evil than giving up the optimization opportunities).

Finally, increasing the size of on-chip cache is about staying away from RAM. Main memory continues to be so much slower than the CPU that it makes sense to put the data closer to the processor— and you can't get much closer than being right on the die. On-die cache sizes have soared, and today most major chip vendors will sell you CPUs that have 2MB of on-board L2 cache. (Of these three major historical approaches to boosting CPU performance, increasing cache is the only one that will continue in the near term.)

Okay. So what does this mean?

A fundamentally important thing to recognize about this list is that all of these areas are concurrency agnostic. Speedups in any of these areas directly lead to speedups in sequential (nonparallel, single-threaded, single-process) applications, as well as applications that do make use of concurrency. That's important because the vast majority of today's applications are single-threaded— and for good reasons.

Of course, compilers have had to keep up; sometimes, you need to recompile your application, and target a specific minimum level of CPU, to benefit from new instructions (MMX, SSE, and the like) and some new CPU features and characteristics. But, by and large, even old applications have always run significantly faster— even without being recompiled to take advantage of all the new instructions and features offered by the latest CPUs.

That world was a nice place to be. Unfortunately, it has already disappeared.

Why You Don't Have 10GHz Today

You can get similar graphs for other chips, but I'm going to use Intel data here. Figure 1 graphs the history of Intel chip introductions by clock speed and number of transistors. The number of transistors continues to climb, at least for now. Clock speed, however, is a different story.

Around the beginning of 2003, you'll note a disturbing sharp turn in the previous trend toward ever-faster CPU clock speeds. I've added lines to show the limit trends in maximum clock speed; instead of continuing on the previous path, as indicated by the thin dotted line, there is a sharp flattening. It has become harder and harder to exploit higher clock speeds due to several physical issues, notably heat (too much of it and too hard to dissipate), power consumption (too high), and current leakage problems.

In short, CPU performance growth as we have known it hit a wall two years ago. Most people have only recently started to notice.

Quick: What's the clock speed on the CPU(s) in your current workstation? Are you running at 10GHz? On Intel chips, we reached 2GHz a long time ago (August 2001), and according to CPU trends before 2003, we now should have the first 10GHz Pentium-family chips. A quick look around shows that, well, actually, we don't. What's more, such chips are not even on the horizon— we have no good idea at all about when we might see them appear.

Well, then, what about 4GHz? We're at 3.4GHz already— surely 4GHz can't be far away? Alas, even 4GHz seems to be remote indeed. In mid-2004, as you probably know, Intel first delayed its planned introduction of a 4GHz chip until 2005, and then in fall

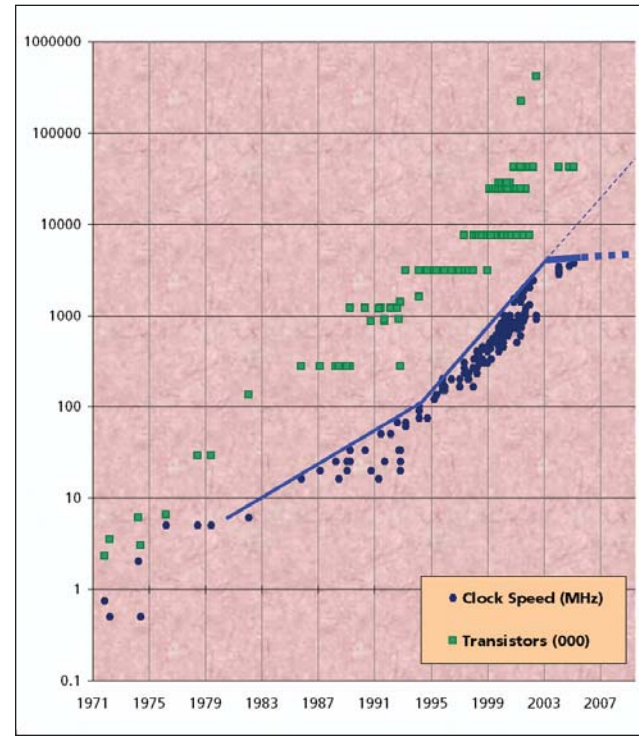


Figure 1: *Intel CPU introductions (sources: Intel, Wikipedia).*

2004, it officially abandoned its 4GHz plans entirely. As of this writing, Intel is planning to ramp up a little further to 3.73GHz early this year (already included in Figure 1 as the upper-rightmost dot), but the clock race really is over, at least for now; Intel's and most processor vendors' futures lie elsewhere as chip companies aggressively pursue the same new multicore directions.

We'll probably see 4GHz CPUs in our mainstream desktop machines someday, but it won't be in 2005. Sure, Intel has samples of their chips running at even higher speeds in the lab—but only by heroic efforts, such as attaching hideously impractical quantities of cooling equipment. You won't have that kind of cooling hardware in your office any day soon, let alone on your lap while computing on the plane.

TANSTAAFL: Moore's Law and The Next Generation(s)

> TANSTAAFL=There ain't no such thing as a free lunch.
>
> —R.A. Heinlein,
> *The Moon Is a Harsh Mistress*

Does this mean Moore's Law is over? Interestingly, the answer in general seems to be "no." Of course, like all exponential progressions, Moore's Law must end someday, but it does not seem to be in danger for a few more years. Despite the wall that chip engineers have hit in juicing up raw clock cycles, transistor counts continue to explode, and it seems CPUs will continue to follow Moore's Law-like throughput gains for some years to come.

The key difference, which is the heart of this article, is that the performance gains are going to be accomplished in fundamentally different ways for at least the next couple of processor generations. And most current applications will no longer benefit from the free ride without significant redesign.

For the near-term future, meaning for the next few years, the performance gains in new chips will be fueled by three main approaches, only one of which is the same as in the past. The near-term future performance growth drivers are:

- Hyperthreading.
- Multicore.
- Cache.

Hyperthreading is about running two or more threads in parallel inside a single CPU. Hyperthreaded CPUs are already available today, and they do allow some instructions to run in parallel. A limiting factor, however, is that although a hyperthreaded CPU has some extra hardware (including extra registers), it still has just one cache, one integer math unit, one FPU, and in general, just one each of most basic CPU features. Hyperthreading is sometimes cited as offering a 5 to 15 percent performance boost for reasonably well-written multithreaded applications, or even as much as 40 percent under ideal conditions for carefully written multithreaded applications. That's good, but it's hardly double, and it doesn't help single-threaded applications.

Multicore is about running two or more actual CPUs on one chip. Some chips, including Sparc and PowerPC, have multicore versions available already. The initial Intel and AMD designs, both due this year, vary in their level of integration but are functionally similar. AMD's seems to have some initial performance design advantages, such as better integration of support functions on the same die; whereas Intel's initial entry basically just glues together two Xeons on a single die. The performance gains should initially be about the same as having a dual-CPU system (only the system will be cheaper because the motherboard doesn't have to have two sockets and associated "glue" chippery), which means something less than double the speed even in the ideal case. Just like today, it will boost reasonably well-written multithreaded applications—not single-threaded ones.

Finally, on-die cache sizes can be expected to continue to grow, at least in the near term. Of these three areas, only this one will broadly benefit most existing applications. The continuing growth in on-die cache sizes is an incredibly important and highly applicable benefit for many applications, simply because space is speed. Accessing main memory is expensive, and you really don't want to touch RAM if you can help it. On today's systems, a cache miss that goes out to main memory typically costs about 10 to 50 times as much as getting the information from the cache; this, incidentally, continues to surprise people because we all think of memory as fast, and it is fast compared to disks and networks, but not compared to on-board cache, which runs at faster speeds. If an application's working set fits into cache, we're golden; and if it doesn't, we're not. That is why increased cache sizes will save some existing applications and breathe life into them for a few more years without requiring significant redesign: As existing applications manipulate more and more data, and as they are incrementally updated to include more code for new features, performance-sensitive operations need to continue to fit into cache. As the Depression-era oldtimers will be quick to remind you, "Cache is king."

(Aside: Here's an anecdote to demonstrate "space is speed" that recently hit my compiler team. The compiler uses the same source base for 32-bit and 64-bit compilers; the code is just compiled as either a 32-bit process or a 64-bit one. The 64-bit compiler gained a great deal of baseline performance by running on a 64-bit CPU, principally because the 64-bit CPU had many more registers to work with and had other code performance features. All well and good. But what about data? Going to 64 bits didn't change the size of most of the data in memory, except that (of course) pointers in particular were now twice the size they were before. As it happens, our compiler uses pointers much more heavily in its internal data structures than most other kinds of applications ever would. Because pointers were now 8 bytes instead of 4 bytes, a pure data size increase, we saw a significant increase in the 64-bit compiler's working set. That bigger working set caused a performance penalty that almost exactly offset the code execution performance increase we'd gained from going to the faster processor with more registers. As of this writing, the 64-bit compiler runs at the same speed as the 32-bit compiler, even though the source base is the same for both and the 64-bit processor offers better raw processing throughput. Space is speed.)

But cache is it. Hyperthreading and multicore CPUs will have nearly no impact on most current applications.

So what does this change in hardware mean for the way we write software? By now, you've probably noticed the basic answer, so let's consider it and its consequences.

What This Means for Software: The Next Revolution

In the 1990s, we learned to grok objects. The revolution in mainstream software development from structured programming to object-oriented programming was the greatest such change in the past 20 years, and arguably in the past 30 years. There have been other changes, including the most recent (and genuinely interesting) nascence of web services, but nothing that most of us have seen during our careers has been as fundamental and as far reaching a change in the way we write software as the object revolution.

Until now. Starting today, the performance lunch isn't free any more. Sure, there will continue to be generally applicable performance gains that everyone can pick up, thanks mainly to cache size improvements. But if you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written, concurrent (usually multithreaded) application. And that's easier said than done, because not all problems are inherently parallelizable and because concurrent programming is hard.

I can hear the howls of protest: "Concurrency? That's not news! People are already writing concurrent applications." That's true. Of a small fraction of developers.

Remember that people have been doing object-oriented programming since at least the days of Simula in the late 1960s. But OOP didn't become a revolution, and dominant in the mainstream, until the 1990s. Why then? The reason the revolution happened was primarily because our industry was driven by requirements to write larger and larger systems that solved larger and larger problems and exploited the greater and greater CPU and storage resources that were becoming available. OOP's strengths in abstraction and dependency management made it a necessity for achieving large-scale software development that is economical, reliable, and repeatable.

Similarly, we've been doing concurrent programming since those same dark ages, writing coroutines and monitors and similar jazzy stuff. And for the past decade or so, we've witnessed incrementally more and more programmers writing concurrent (multithreaded, multiprocess) systems. But an actual revolution marked by a major turning point toward concurrency has been slow to materialize. Today, the vast majority of applications are single-threaded, and for good reason.

By the way, on the matter of hype: People have always been quick to announce "the next software development revolution," usually about their own brand-new technology. Don't believe it. New technologies are often genuinely interesting and sometimes beneficial, but the biggest revolutions in the way we write software generally come from technologies that have already been around for some years and have already experienced gradual growth before they transition to explosive growth. This is necessary: You can only base a software development revolution on a technology that's mature enough to build on (including having solid vendor and tool support), and it generally takes any new software technology at least seven years before it's solid enough to be broadly usable without performance cliffs and other gotchas. As a result, true software development revolutions like OOP happen around technologies that have already been undergoing refinement for years, often decades. Even in Hollywood, most genuine "overnight successes" have really been performing for many years before their big break.

Concurrency is the next major revolution in how we write software. Different experts still have different opinions on whether it will be bigger than OO, but that kind of conversation is best left to pundits. For technologists, the interesting thing is that concurrency is of the same order as OOP both in the (expected) scale of the revolution and in the complexity and learning curve of the technology.

Benefits and Costs of Concurrency

There are two major reasons for which concurrency, especially multithreading, is already used in mainstream software. The first is to logically separate naturally independent control flows; for example, in a database replication server I designed, it was natural to put each replication session on its own thread because each session worked completely independently of any others that might be active (as long as they weren't working on the same database row). The second and less common reason to write concurrent code in the past has been for performance, either to scalably take advantage of multiple physical CPUs or to easily take advantage of latency in other parts of the application; in my database replication server, this factor applied as well, and the separate threads were able to scale well on multiple CPUs as our server handled more and more concurrent replication sessions with many other servers.

There are, however, real costs to concurrency.

Some of the obvious costs are actually relatively unimportant. For example, yes, locks can be expensive to acquire, but when

used judiciously and properly, you gain much more from the concurrent execution than you lose on the synchronization, if you can find a sensible way to parallelize the operation and minimize or eliminate shared state.

Perhaps the second-greatest cost of concurrency is that not all applications are amenable to parallelization.

Probably the greatest cost of concurrency is that concurrency really is hard: The programming model, meaning the model in programmers' heads that they need to reason reliably about their program, is much harder than it is for sequential control flow.

Everybody who learns concurrency thinks he understands it, but ends up finding mysterious races he thought weren't possible and discovers that he didn't actually understand it after all. As developers learn to reason about concurrency, they find that usually those races can be caught by reasonable in-house testing, and they reach a new plateau of knowledge and comfort. What usually doesn't get caught in testing, however, except in shops that understand why and how to do real stress testing, are those latent concurrency bugs that surface only on true multiprocessor systems, where the threads aren't just being switched around on a single processor but really do execute truly simultaneously and thus expose new classes of errors. This is the next jolt for developers who thought that, surely now, they know how to write concurrent code: I've come across many teams whose application worked fine even under heavy and extended stress testing, and ran perfectly at many customer sites, until the day that a customer actually had a real multiprocessor machine — and then deeply mysterious races and corruptions started to manifest intermittently. In the context of today's CPU landscape, then, redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end — going straight to the least forgiving, truly parallel environment that is most likely to expose the things you got wrong. Even when you have a team that can reliably write safe concurrent code, there are other pitfalls; for example, concurrent code that is completely safe but isn't any faster than it was on a single-core machine, typically because the threads aren't independent enough and share a dependency on a single resource that reserializes the program's execution. This stuff gets pretty subtle.

Just as it is a leap for a structured programmer to learn OOP ("what's an object?" "what's a virtual function?" "how should I use inheritance?" and beyond the "whats" and "hows," "why are the correct design practices actually correct?"), it's a leap of about the same magnitude for a sequential programmer to learn concurrency ("what's a race?" "what's a deadlock?" "how can it come up, and how do I avoid it?" "what constructs actually serialize the program that I thought was parallel?" and beyond the "whats" and "hows," "why are the correct design practices actually correct?"). The vast majority of programmers aren't there today, just as the vast majority of programmers 15 years ago didn't yet grok objects. But the concurrent programming model is learnable, particularly if we stick to lock-based programming, and once grokked, it isn't that much harder than OOP and hopefully can become just as natural. Just be ready and allow for the investment in training and time, for you and for your team.

(I deliberately limit the aforementioned discussion to lock-based concurrent programming models. There is also lock-free programming, supported most directly at the language level in Java 5 and in at least one popular C++ compiler. But concurrent lock-free programming is known to be very much harder for programmers to understand and reason about than even concurrent lock-based programming. Most of the time, only systems and library writers should have to understand lock-free programming, although virtually everybody should be able to take advantage of the lock-free systems and libraries those people produce.)

What it Means for Us

Okay, back to what it means for us.

- The clear primary consequence we've already covered is that applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains that have now started becoming available and will continue to materialize over the next several years. "Oh, performance doesn't matter so much, computers just keep getting faster" has always been a naïve statement to be viewed with suspicion, and for the near future, it will almost always be simply wrong.

  Now, not all applications (or more precisely, important operations of an application) are amenable to parallelization. Some problems, such as compilation, are almost ideally parallelizable. Others aren't; the usual counterexample here is that just because it takes one woman nine months to produce a baby doesn't imply that nine women could produce one

Myths and Realities: 2×3GHz≠6GHz

So, a dual-core CPU that combines two 3GHz cores practically offers 6GHz of processing power. Right? Wrong. Even having two threads running on two physical processors doesn't mean getting two times the performance. True, there are some kinds of problems that are inherently parallelizable and can approach linear throughput gains; a typical example is compilation, which can run close to twice as fast on a carefully managed dual-CPU dual-disk system.

Similarly, most multithreaded applications won't run twice as fast on a dual-core box, although they should run faster than on a single-core CPU. The performance gain just isn't linear, that's all.

Why not? First, there is coordination overhead between the cores to ensure cache coherency (a consistent view of cache and of main memory) and to perform other handshaking. Today, a two- or four-processor machine isn't really two or four times as fast as a single CPU even for multithreaded applications. The problem remains essentially the same even when the CPUs in question sit on the same die.

Second, unless the two cores are running different processes, or different threads of a single process that are well-written to run independently and almost never wait for each other, they won't be well utilized. (Despite this, I will speculate that today's single-threaded applications as actually used in the field could see a performance boost for most users by going to a dual-core chip, not because the extra core is actually doing anything useful, but because it is running the adware and spyware that infest many users' systems and are otherwise slowing down the single CPU that user has today. I leave it up to you to decide whether adding a CPU to run your spyware is the best solution to that problem.)

If you're running a single-threaded application, then the application can only make use of one core. There should be some speedup as the operating system and the application can run on separate cores, but typically, the OS isn't going to be maxing out the CPU anyway, so one of the cores will be mostly idle. (Again, the spyware can share the OS's core most of the time.)

—H.S.

(continued from page 20)

baby in one month. You've probably come across that analogy before. But did you notice the problem with leaving the analogy at that? Here's the trick question to ask the next person who uses it on you: Can you conclude from this that the Human Baby Problem is inherently not amenable to parallelization? Usually, people relating this analogy err in quickly concluding that it demonstrates an inherently nonparallel problem, but that's actually not necessarily correct at all. It is indeed an inherently nonparallel problem if the goal is to produce one child. It is actually an ideally parallelizable problem if the goal is to produce many children! Knowing the real goals can make all the difference. This basic goal-oriented principle is something to keep in mind when considering whether and how to parallelize your software.

- Perhaps a less obvious consequence is that applications are likely to become increasingly CPU-bound. Of course, not every application operation will be CPU-bound, and even those that will be affected won't become CPU-bound overnight if they aren't already, but we seem to have reached the end of the "applications are increasingly I/O-bound or network-bound or database-bound" trend, because performance in those areas is still improving rapidly (gigabit Wi-Fi, anyone?) while traditional CPU performance-enhancing techniques have maxed out. Consider: We're stopping in the 3GHz range for now. Therefore, single-threaded programs are likely not going to get much faster any more for now except for benefits from further cache size growth (which is the main good news). Other gains are likely to be incremental and much smaller than we've been used to seeing in the past; for example, as chip designers find new ways to keep pipelines full and avoid stalls, which are areas where the low-hanging fruit has already been harvested. The demand for new application features is unlikely to abate, and even more so the demand to handle vastly growing quantities of application data is unlikely to stop accelerating. As we continue to demand that programs do more, they will increasingly often find that they run out of CPU to do it unless they can code for concurrency.

There are two ways to deal with this sea change toward concurrency. One is to redesign your applications for concurrency. The other is frugality, or writing code that is more efficient and less wasteful. This leads to the third interesting consequence:

- Efficiency and performance optimization will get more — not less — important. Those languages that already lend themselves to heavy optimization will find new life; those that don't will need to find ways to compete and become more efficient and optimizable. Expect long-term increased demand for performance-oriented languages and systems.
- Finally, programming languages and systems will increasingly be forced to deal well with concurrency. Java has included support for concurrency since its beginning, although mistakes were made that later had to be corrected over several releases to do concurrent programming more correctly and efficiently. C++ has long been used to write heavy-duty multithreaded systems well, but it has no standardized support for concurrency at all (the ISO C++ standard doesn't even mention threads, and does so intentionally), and so typically, the concurrency is of necessity accomplished by using nonportable platform-specific concurrency features and libraries. (It's also often incomplete; for example, static variables must be initialized only once, which typically requires that the compiler wrap them with a lock, but many C++ implementations do not generate the lock.) Finally, there are a few concurrency standards, including pthreads and OpenMP, and some of these support implicit as well as explicit parallelization. Having the compiler look at your single-threaded program and automatically figure out how to parallelize it implicitly is fine and dandy, but those automatic transformation tools are limited and don't yield nearly the gains of explicit concurrency control that you code yourself.

Conclusion

If you haven't done so already, now is the time to take a hard look at the design of your application, determine what operations are CPU-sensitive now or are likely to become so soon, and identify how those places could benefit from concurrency. Now is also the time for you and your team to grok concurrent programming's requirements, pitfalls, styles, and idioms.

A few rare classes of applications are naturally parallelizable, but most aren't. Even when you know exactly where you're CPU-bound, you may well find it difficult to figure out how to parallelize those operations; all the more reason to start thinking about it now. Implicitly parallelizing compilers can help a little, but don't expect much; they can't do nearly as good a job of parallelizing your sequential program as you could do by turning it into an explicitly parallel and threaded version.

Thanks to continued cache growth and probably a few more incremental straight-line control flow optimizations, the free lunch will continue a little while longer; but starting today, the buffet will only be serving that one entrée and that one dessert. The filet mignon of throughput gains is still on the menu, but now it costs extra-extra development effort, extra code complexity, and extra testing effort. The good news is that for many classes of applications the extra effort will be worthwhile because it will let them fully exploit the continuing exponential gains in processor throughput.

DDJ

64-Bit Computing & JVM Performance

More horsepower doesn't always mean better performance

SERGIY KYRYLKOV

The era of 64-bit computing started with the release of the Alpha processor by Digital Equipment Corp. in 1992. Later, several other major computer hardware companies, namely Hewlett-Packard, Fujitsu, Sun Microsystems, and IBM, moved into the 64-bit market with their offerings. In 1995, the Fujitsu-owned HAL Computer launched the industry's first workstations based on a 64-bit SPARC CPU, SPARC64. Shortly after HAL's announcement, Sun launched the long expected Ultra 1 and 2 workstations, which used Sun's 64-bit UltraSPARC processor. In 1997, IBM released RS64, the first 64-bit PowerPC RISC chip. In 1998, IBM supplemented RS64 with a 64-bit SMP chip, POWER3. It took another five years for the 64-bit computing to come to mass market.

In this article, I examine two modern 64-bit platforms widely available in the sub-$5000 range—the AMD64 and PowerPC64. In the process, I evaluate the performance of 32- and 64-bit Java Virtual Machines from two major JVM vendors, Sun Microsystems and IBM, using the SPECjvm98 and SPECjbb2000 benchmarks from Standard Performance Evaluation Corp. (http://www.spec.org/).

The AMD64 is a 64-bit platform from Advanced Micro Devices (AMD) that extends the industry-standard x86 instruction set architecture. It was designed to deliver full compatibility with existing x86 applications and operating systems without paying performance penalties when working in 32-bit mode. In April 2003, AMD announced the availability of Opteron, the first processor supporting AMD64 architecture.

In 2003, IBM introduced PowerPC 970, a single-core processor that was derived from IBM's POWER4 dual-core CPU and brought processing power of the 64-bit PowerPC architecture to desktops and low-end servers. Additionally, the PowerPC 970, like the POWER4, was also able to process 32-bit instructions natively without any performance penalty. Soon after this, in August 2003, Apple Computer started shipping Power Mac G5 computers featuring PowerPC 970.

Java Backgrounder

Around the time when the first 64-bit processor came to life, the history of Java technology started. Java is a robust, general-purpose, object-oriented, architecture-neutral, portable, secure, multithreaded programming language, with implicit memory management. Java's object-oriented features are mostly the same as C++, with the addition of interfaces and extensions for more dynamic method resolution. Unlike C++, Java does not support operator overloading, multiple inheritance, or automatic type coercion. Robustness is mostly achieved by extensive dynamic (runtime) checking and a built-in exception-handling mechanism. The Java compiler generates bytecode instructions that are independent of any specific architecture, and thus provides architecture neutrality. Additional portability is achieved by specifying the sizes of the primitive data types and the behavior of arithmetic operators on these types. For example, *int* always means a signed two's complement 32-bit integer, and *float* always means a 32-bit IEEE 754 floating-point number. Java also has a set of synchronization primitives that are based on the widely used monitor and condition variable paradigm. Automatic garbage collection (GC) simplifies the task of Java programming and dramatically decreases the number of bugs, but makes the system somewhat more complicated.

In 1991, one year before Digital Equipment Corp. introduced the first 64-bit processor, Sun Microsystems initiated "the Green Project." The goal of the project was to anticipate and plan for the "next wave" of computing. The initial conclusion of the project was that the world would soon see the fusion of mobile digital devices and computers. In the summer of 1992, the Green Team presented *7 ("StarSeven"), a working demo of an interactive handheld entertainment device controller with an animated touch-screen user interface. The device was able to control a number of different platforms by using Oak, an entirely new programming language. The main feature of Oak, developed by James Gosling, was that it was a completely processor-independent language. During the next several years, the language was retargeted for the Internet and later became known as "Java." The name "Oak" was dismissed because of the copyright issues.

> "The benefits of 64-bit computing show up in a number of applications"

In May 1995, Sun formally announced Java, a programming language that lets developers write a program once and run it on multiple operating systems and hardware platforms ("write once, run anywhere"). In 1996, Sun released the Java Development Kit (JDK 1.0), and shortly thereafter, 10 major operating-system vendors announced their plan to distribute Java technology with their products—including Microsoft, which licensed Java from Sun for five years at a cost of approximately $3.75 million per year. In October 1996, Sun announced the first Just-in-Time compiler for the Java platform. JDK 1.1, shipped in February 1997, was downloaded more than 220,000 times within the next three weeks after its release. By the beginning of the next year, this number reached 2 million.

Sergiy is CTO of SA Consulting and can be contacted at mail@sergiy.kyrylkov.name.

At the end of 1998, the Java 2 platform was released. Roughly half a year later, in the middle of 1999, Sun announced the three editions of the Java platform: J2ME (Java 2 Micro Edition), for mobile, wireless, and other resource-constrained environments; J2SE (Java 2 Standard Edition), for desktop environments; and J2EE (Java 2 Enterprise Edition), for Java-based application servers. J2EE laid out a framework for a number of Java development technologies that have already gained widespread use, such as Enterprise JavaBeans (EJB) and JavaServer Pages (JSP). The next upgrade to Java technology, J2SE 1.3, appeared in May 2000 and several weeks later it gained industry support from Apple with Mac OS X.

J2SE 1.4 was released in February 2002 and was a major new release of the Java platform. It contained 62 percent more classes and interfaces than J2SE 13. Among other features, it provided extensive XML support, support for secure sockets (using the SSL and TLS protocols), new I/O API, logging, regular expressions, and assertions.

In September 2004, the most recent release of Java, J2SE 5.0 (internal version number 1.5.0) codename "Tiger," became publicly available. Tiger contains the first significant updates to Java since its 1.0 release in 1996, including support for generics, autoboxing and autoboxing of primitive types, enhanced *for* loops, enumerated types, formatted I/O, and *varargs*.

The Java Virtual Machine (JVM) is a specification for software responsible for running Java programs compiled into a special instruction set—Java bytecode. The JVM is an abstract computing machine and is responsible for Java hardware and OS independence, the small size of Java compiled code, and has the ability to prevent malicious programs from executing. The Java Virtual Machine does not assume any particular implementation technology, hardware, or operating system. There are several JVM components, whose 32-bit and 64-bit version performance may differ, adding to the general performance difference between 32-bit and 64-bit binaries. Among other things, for example, they include the Just-in-Time (JIT) compiler and garbage collection (GC).

The JIT compiler has been a part of JVM since JDK 1.0.2, when Java was viewed only as a client-side technology. The JIT compiler implements dynamic translation of Java bytecode to hardware machine code before execution. The idea behind JIT is that Java bytecode is smaller and easier to compile than the original Java source code. The result is that the time spent compiling Java bytecode on any platform to machine code is much less than the time to compile machine code straight from the Java source. In addition, JITed code can run as fast as statically compiled code. In 32-bit and 64-bit JVMs, the corresponding JITs take somewhat different time to compile Java bytecode to the final actual machine code and can apply different optimizations, affecting the total performance difference between the two versions both in client- and server-side applications.

Garbage collection is an automatic memory-management system, which reclaims memory no longer needed by objects. From the point of view of software engineering, this provides one of the biggest advantages of Java—programmers can forget about low-level memory-management details. Garbage collection also removes the two big sources of bugs: incomplete deallocation (memory leaks) and premature deallocation (corrupted pointers). Garbage collection accounts for a significant portion of the running time of the Java application, since it has to be performed regularly to free the Java heap of inaccessible objects. Since the size of data in the Java heap in 32-bit and 64-bit differs in one way or another, the garbage collection performance difference also contributes to the general performance difference between 32-bit and 64-bit JVMs.

64-Bit Backgrounder

64-bit computing comprises several key elements, most importantly 64-bit addressing. In practical terms, 64-bit addressing is achieved with 64-bit integer registers (or general-purpose registers in RISC terms). 64-bit registers let 64-bit pointers fit into a single register. The advantage of 64-bit pointers is that they make it possible to address huge (as for present times) amounts of memory. While a 32-bit processor is capable of utilizing only 2^{32} bytes or about 4GB of memory, a 64-bit processor theoretically can address 2^{64} bytes or about 18 billion GB of memory. The practical limit of addressable memory in modern 64-bit systems is usually lower, depending on specific hardware architecture and operating systems. For example, in Linux-based operating systems, the addressable memory is limited to 2^{42} bytes or 4096GB due to the current design of internal Linux kernel data structures. Obviously, this is still good enough to break space limitations of the current 32-bit systems.

The second important aspect of 64-bit computing is 64-bit integer arithmetic. Again, this is a simple consequence of having 64-bit wide integer registers capable of storing much larger integer quantities. The direct result of this may be a significant performance improvement for certain types of applications dealing with intensive integer computations of large data.

The third, but equally important characteristic of 64-bit computing is the use of 64-bit operating systems and applications. Such software must support all 64-bit features of the hardware, including 64-bit addressing and arithmetic. Usually it comes with some extra benefits, such as the ability to operate on more files and larger files (although this also may be a feature of certain 32-bit software).

The benefits of 64-bit computing show up in a number of applications. Database servers use a large address space for scalability, maintaining larger buffer pools, caches, and sort heaps in memory to reduce the volume of I/O they perform. They can also allocate more per-user memory, support many more users, and work with much larger files. Simulation and other computationally intensive programs benefit from keeping much larger arrays of data entirely in memory. Finally, a large group of Java programs—J2EE application servers—have been enjoying the benefits of 64-bit computing for some time now, utilizing modern 64-bit Java implementations.

The major drawback of 64-bit computing comes from the fact that 64-bit binaries are typically larger than their 32-bit counterparts. As a consequence, with a larger final machine code size and a given size of cache and translation lookaside buffer (TLB), the chances of both cache and TLB misses increase. This, in turn, may decrease the performance and negate the 64-bit benefits.

Performance Evaluation

The test systems I used to examine the performance of the 32- and 64-bit JVMs from Sun and IBM are two 64-bit dual-CPU workstations—an AMD64-based Opteron system and the PowerPC64-based Apple Power Mac G5. Both workstations run Linux-based 64-bit operating systems, Fedora Core 2, and a beta version of Terra Soft Solutions Y-HPC accordingly, featuring Linux kernel 2.6.x.

I used the SPECjvm98 and SPECjbb2000 benchmarks from SPEC to gauge the performance of the JVMs. SPECjvm98 measures the client-side performance of Java Virtual Machines using these seven applications:

- _201_compress, a popular compression program.
- _202_ jess, a Java version of NASA's CLIPS rule-based expert system.
- _209_db, data management benchmarking software.
- _213_ javac, the JDK Java compiler.
- _222_mpegaudio, an MPEG-3 audio decoder.
- _227_mtrt, a dual-threaded program that ray traces an image file.
- _228_ jack, a real parser-generator.

SPECjbb2000 (the Java Business Benchmark) is a benchmark for evaluating the

performance of server-side Java, which emulates a three-tier system, a common type of server-side Java applications.

With the SPECjvm98 benchmark, I measured the total execution time of every benchmark application in seconds, where the lower value is better. The heap size for all SPECjvm98 benchmarks is varied by the JVM between a minimum heap size of 16MB and maximum heap size of 32MB. In the SPECjbb2000 benchmark, I measured the number of operations per second for three different heap sizes. Here, the higher value corresponds to the higher performance. Each benchmark application was run three times. For final results, the best runs are reported.

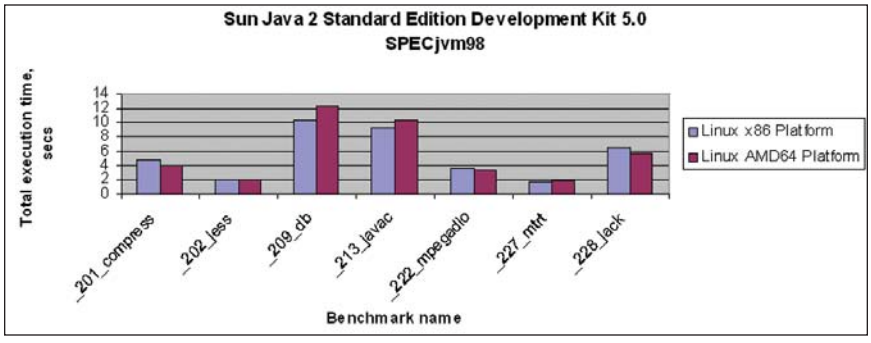


Figure 1: *Performance of 32-bit and 64-bit Linux versions of the Sun Java 2 Standard Edition Development Kit 5.0 (J2SE 1.5.0) in SPECjvm98 benchmarks on the AMD64 platform.*

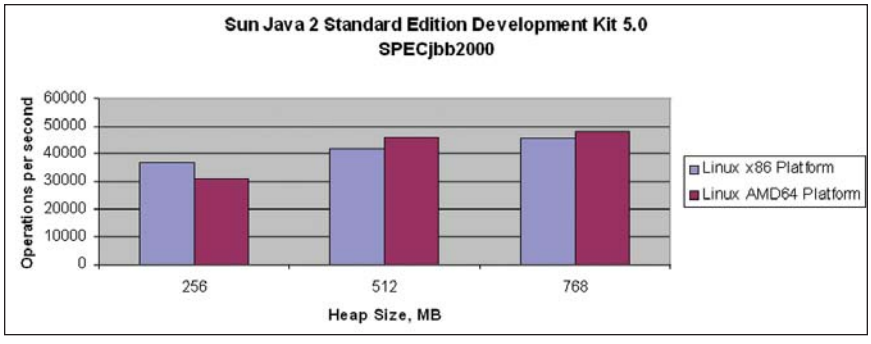


Figure 2: *Performance of 32-bit and 64-bit Linux versions of Sun Java 2 Standard Edition Development Kit 5.0 (J2SE 1.5.0) in SPECjbb2000 benchmarks on the AMD64 platform.*

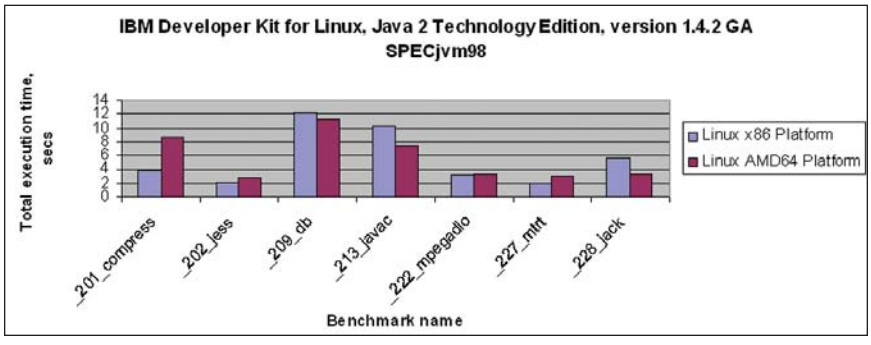


Figure 3: *Performance of 32-bit and 64-bit versions of IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjvm98 benchmarks on the AMD64 platform.*

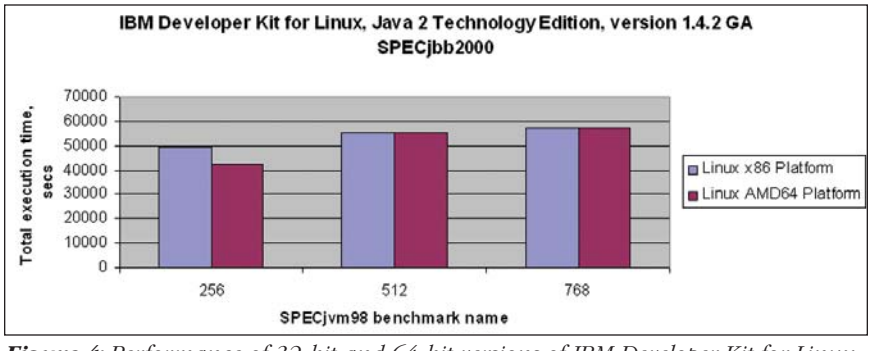


Figure 4: *Performance of 32-bit and 64-bit versions of IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjbb2000 benchmarks on the AMD64 platform.*

Figures 1 and 2 show the performance of 32-bit and 64-bit Linux versions of the Sun Java 2 Standard Edition Development Kit 5.0 (J2SE 1.5.0) in SPECjvm98 and SPECjbb2000 benchmarks on the AMD64 platform. In three SPECjvm98 benchmark applications out of seven—_201_compress, _222_mpegaudio, and _228_ jack—the 64-bit version of the JVM shows a better performance than its 32-bit counterpart. In SPECjbb2000, the performance of the 64-bit version is higher only for large enough heap sizes. In the case of 256MB heap size, the fact that the total amount of live data is larger in the 64-bit version of the JVM causes more frequent garbage collections, which decreases the application throughput.

Figures 3 and 4 show the performance of 32-bit and 64-bit versions of the IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjvm98 and SPECjbb2000 benchmarks on the AMD64 platform. For this JVM, a different set of three out of seven benchmark applications—_209_db, _213_ javac, and again _228_ jack—shows better performance in the 64-bit environment. In the SPECjbb2000 benchmark, the 64-bit version of IBM's JVM does not show better performance than its 32-bit counterpart in any of the three tested heap sizes.

Figures 5 and 6 illustrate the performance of 32-bit and 64-bit versions of the IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjvm98 and SPECjbb2000 benchmarks on the PowerPC64 platform. Here, in all SPECjvm98 benchmark applications and in SPECjbb2000, performance of the 64-bit JVM is worse than the performance of its 32-bit counterpart.

Conclusion

Based on the benchmark results for the PowerPC64 platform running a Linux-based operating system (the only 64-bit OS available for Apple Power Mac G5 today) and IBM JVM, you can conclude that on this platform, any application that does not require 64-bit features should be used on a 32-bit JVM because, in all cases, performance of the 64-bit JVM here is lower than the performance of its 32-bit counterpart.

On the AMD64 platform running a Linux-based OS and both Sun and IBM JVMs, you see that (in the general case) it is hard to predict performance difference for Java apps run on the 32-bit JVMs and their 64-bit counterparts. The performance benefits here are both application and JVM dependent. In a case when maximum performance is required, it is necessary to benchmark a specific application in its specific execution environment to be able to evaluate potential benefits of switching to 64-bit computing.

There are several important things worth noting here. First, although both SPECjvm98 and SPECjbb2000 are industry-standard benchmarks, they are limited in scope. Thus, the obtained results may hold for a large set of Java applications, but not for the whole range. Second, with more and more widespread adoption of 64-bit computing, we can expect continuous improvement of the 64-bit tools, including the 64-bit JVMs, which may further improve their performance. Third, we tested only several specific combinations of hardware platforms, operating systems, and JVMs. Thus, the results of the JVM benchmarks on the 64-bit edition of Windows XP/2003 or the upcoming 64-bit Mac OS X may provide quite different insights.

DDJ

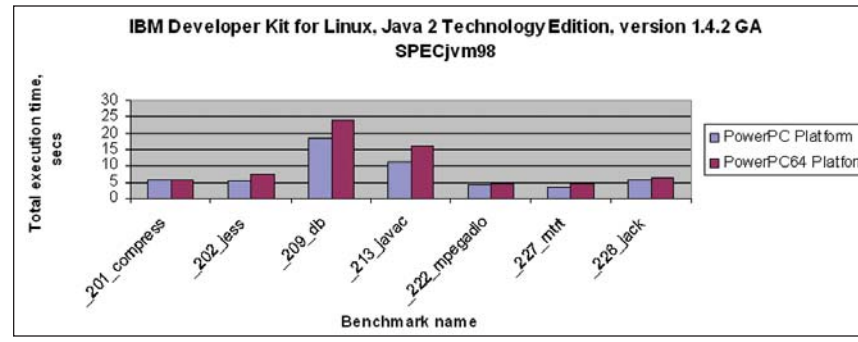


Figure 5: *Performance of 32-bit and 64-bit versions of IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjvm98 benchmarks on the PowerPC64 platform.*

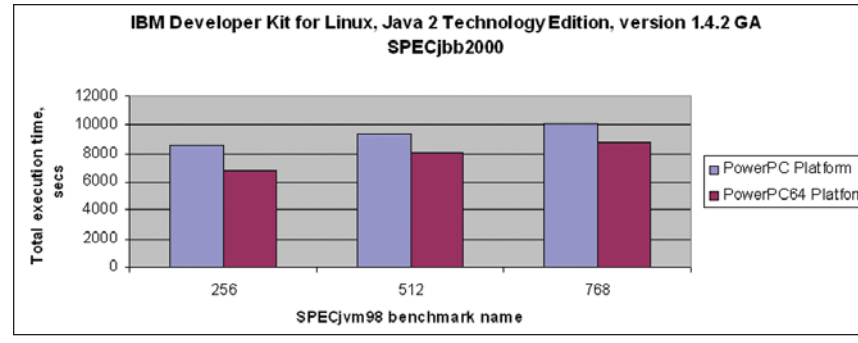


Figure 6: *Performance of 32-bit and 64-bit versions of IBM Developer Kit for Linux, Java 2 Technology Edition, Version 1.4.2 GA in SPECjbb2000 benchmarks on the PowerPC64 platform.*

Windows & the World of 64-Bit Computing

Greater performance and more memory

VIKRAM SRIVATSA

64-bit computing is moving into the mainstream and will gradually replace 32-bit computing. This shift will have a major impact on software in its current form. Among other things, the shift will require porting applications and rewriting system software, including the operating system. In this article, I examine the structure of what will likely be major players in the 64-bit software world—64-bit Windows and the 64-bit version of the Common Language Runtime (CLR). Along the way, I point out some of the advantages of moving to 64-bit platforms.

While 64-bit processors have been around for some time, their adoption has been gradual, mainly due to the lack of software to run on them. To take full advantage of 64-bit processors, the software needs to be built for 64-bit microprocessors—this cannot happen overnight. More recently, however, 64-bit processors are picking up momentum because of a combined effort on the part of both software and hardware vendors.

Early last year, for instance, we saw the arrival of 64-bit processors from Intel and Advanced Micro Devices (AMD)—the Intel Itanium based on IA-64 architecture from Intel, and AMD Opteron and AMD Athlon64 based on x86-64 architecture from AMD, respectively. Moreover, additional developments have occurred since the beginning of last year. For one thing, AMD has emerged a leader in 64-bit microprocessor sales. Second, Hewlett-Packard has also embraced the AMD processors and the AMD Opteron-based HP ProLiant Servers are now available. Also, Intel has announced its own equivalent of x86-64 in the form of Intel EM64T (Extended Memory 64 Technology).

Microsoft Windows 64-Bit Edition

On the software side, Microsoft has been working on a 64-bit version of Windows—Windows XP Professional x64 Edition for the desktop (http://www.microsoft.com/windowsxp/64bit/evaluation/upgrade.mspx), and Windows Server 2003 x64 Edition and Windows Server 2003 for servers (http://www.microsoft.com/windowsserver2003/64bit/x64/trial/default.mspx).

The advantages of 64-bit Windows over 32-bit Windows include an increase in performance and scalability (since the 64-bit processor is capable of processing more data per clock cycle), faster performance and better accuracy of numeric calculations, and the capability to address more memory. Addressing more memory means that a single machine can support more users than its 32-bit counterpart. This means that the total cost of ownership reduces because a single machine supports more users and more applications than before, which reduces the number of servers required for an organization to run its business.

> "For Windows 64-bit to be successful, it needs to ensure that current 32-bit applications are supported"

However, for Windows 64-bit to be successful, it needs to ensure that current 32-bit applications are supported. Consequently, the migration from 32-bit to 64-bit

Vikram is a software designer/specialist for Hewlett-Packard GDIC. He can be contacted at vikram404@gmail.com.

will take time, during which both 32-bit and 64-bit applications need to work side by side. To support this shift, Windows 64-bit edition includes a subsystem known as "WOW64."

WOW64

WOW64, short for "Windows-32-on-Windows-64," is responsible for providing two levels of support for 32-bit legacy applications.

First, the system files in Windows x64 Edition are not present on just the Windows\System32 folder, but split into two folders to separate the 32-bit applications from the 64-bit applications. The WOW64 subsystem intercepts calls from a 32-bit legacy application and redirects it to the Windows\SysWow64 folder; see Figure 1. If the call is from a 64-bit application, then the call is routed to the Windows\System32 folder and does not involve the WOW64. What's notable here is that Microsoft has retained the name System32 for the folder, which hosts the 64-bit system files. Figure 2, a snapshot from a system running Windows Server 2003 x64 Edition, highlights the classification of the Program Files folder into Program Files, which stores 64-bit applications and Program Files(x86), which stores 32-bit legacy applications.

Second, the WOW64 subsystem also provides redirection at the Registry level; see Figure 3. If the call is from a 32-bit application, then the call to access the HKLM\Software registry hive is intercepted by the WOW64 subsystem and redirected to the HKLM\Software\Wow6432Node. If the call is from a 64-bit application, then it is routed to the HKLM\Software node. Figure 4, the Registry from a system running Windows 2003 Server x64 Edition, shows the Wow6432Node.

Although the compatibility has been achieved with respect to 32-bit applications, the same is not true regarding device drivers. The 64-bit edition requires 64-bit native drivers for all devices that are part of the system.

64-Bit Common Language Runtime

For 64-bit platforms to gain widespread acceptance, there must be widespread availability of developer tools and developer platforms. Microsoft's approach has been to add 64-bit support for its core development platform—the .NET Framework.

The .NET Framework 2.0 is currently in Beta 1 and codenamed "Whidbey." This release has two versions of the Framework— one for 32-bit applications and one for 64-bit applications (http://msdn.microsoft.com/netframework/downloads/updates/default.aspx) means that the 64-bit edition of Windows will have two copies of the runtime. The .NET Framework coupled with Visual Studio 2005 provides a platform for developing 64-bit applications. The 32-bit version of the .NET Framework will reside in the \Windows\Microsoft.NET\Framework folder, while the 64-bit version of the .NET Framework resides in the \Windows\Microsoft.NET\Framework64 Folder; see Figure 5. The configuration options for these two versions of the Framework are also listed separately in the Administrative Tools Menu; see Figure 6.

Why two frameworks? One of the widely claimed advantages of compiling to MSIL is that the Just-In-Time compilation takes care of hardware-related specifics. In this case, however, there are other factors that have to be considered, such as PInvoke (Platform Invocation Services) and COM Interop, which need special handling. It is also possible to write assemblies using Visual C++ .NET, which contains both managed and unmanaged sections. Such assemblies are referred to as "mixed-mode" assemblies or "IJW" assemblies, where IJW stands for "It Just Works." Whenever such scenarios are involved, there needs to be platform-specific code; hence the need for two frameworks, each specific to the particular platform that arises. Consequently, Microsoft ships two versions of the Framework. This concept becomes clearer when you consider the Global Assembly Cache (Figure 7). The key column to look at in Figure 7 is the one entitled "Processor Architecture," of which there are three types—x86, AMD64, and MSIL. "AMD64" is shown since this snapshot is from a machine running Windows x64 Edition. In the case of Intel Itanium-based systems, the Processor Architecture "Itanium" would replace AMD64. Processor architecture denotes the platform for which the assembly has been built.

There is only a single copy of the assemblies compiled to MSIL because MSIL is neutral to processor architecture and the same assembly works on either x86 or the AMD64 platforms without modifications. These MSIL assemblies are also referred to as "portable assemblies." For example, the System.Xml in Figure 7 has only one copy of the System.Xml, which has the Processor Architecture of MSIL.

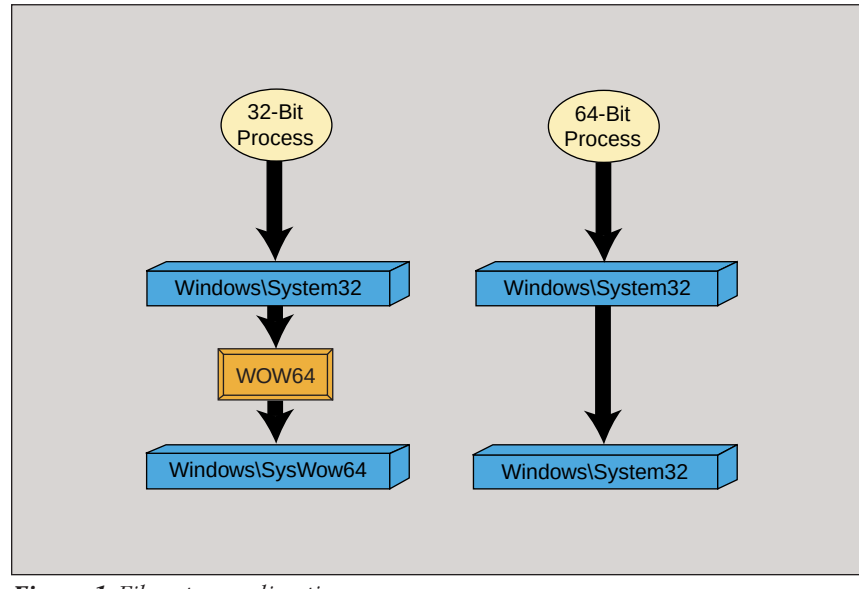


Figure 1: *Filesystem redirection.*

Figure 2: *A system running Windows Server 2003 x64 Edition.*

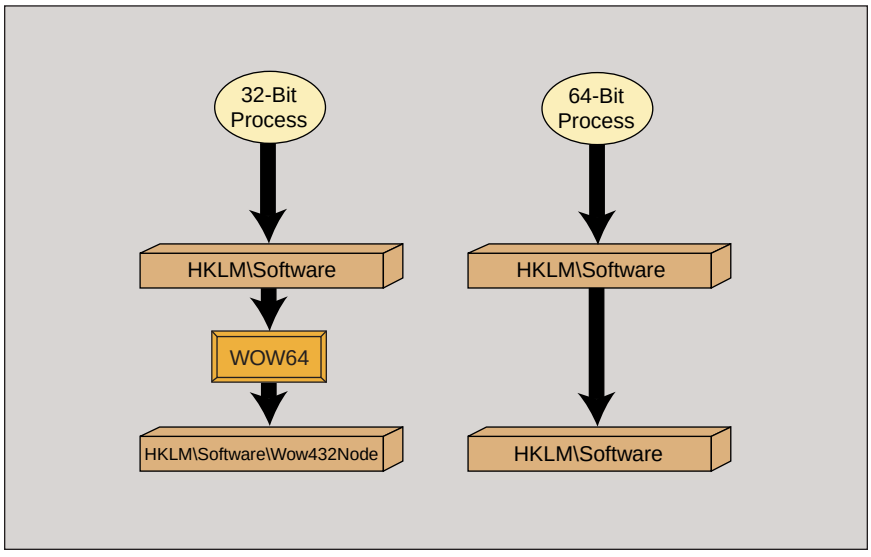


Figure 3: *Registry redirection.*

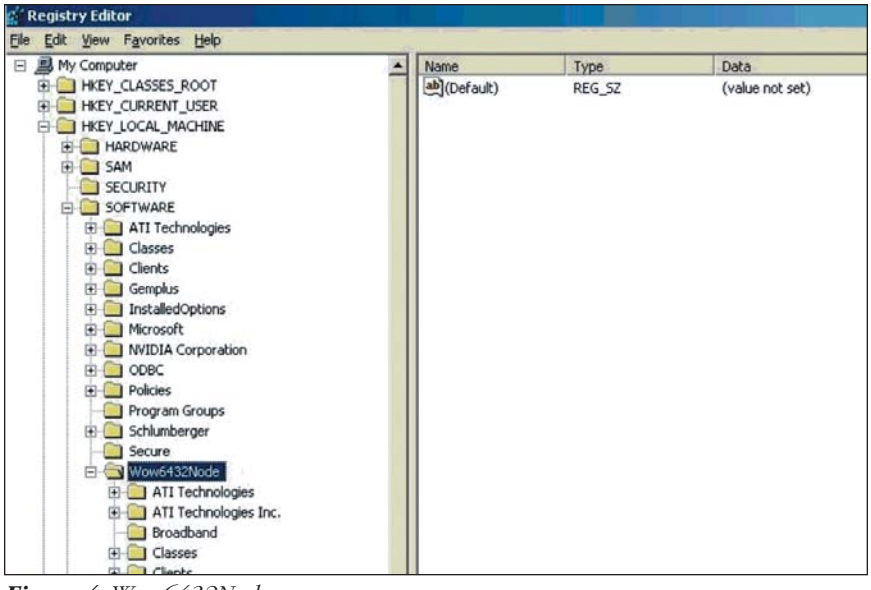


Figure 4: *Wow6432Node.*

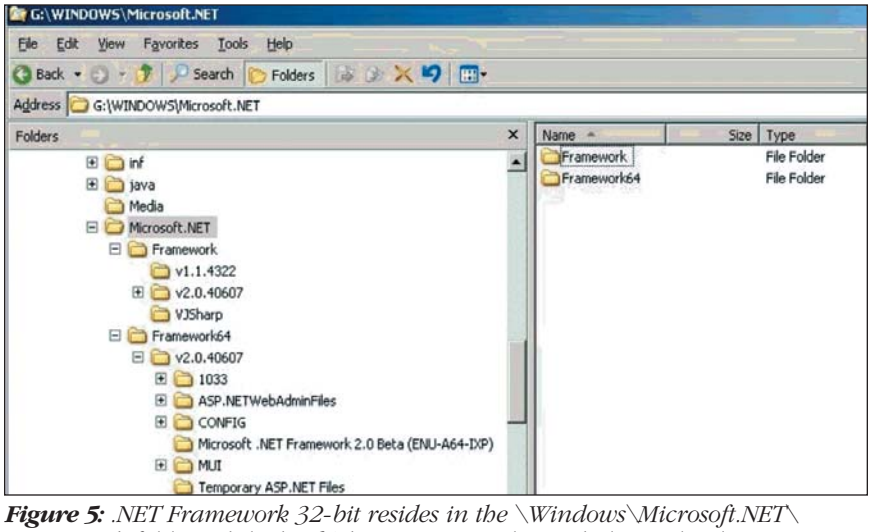


Figure 5: *.NET Framework 32-bit resides in the \Windows\Microsoft.NET\Framework folder, while the 64-bit version resides in the \Windows\Microsoft.NET\Framework64 folder.*

However, assemblies that are built targeting particular processor architectures (such as x86 architecture or the AMD64 architecture) need to be present separately, with one assembly built specifically for the AMD64 architecture and one assembly built specifically for the x86 architecture. These assemblies are referred to as "platform-specific assemblies." For example, take a look at the "System.EnterpriseServices" assembly in Figure 7. There is a separate assembly for AMD64 and x86.

The Whidbey team has tried to have as many MSIL-based assemblies as possible, so that there is only one copy on disk. However, in some scenarios, it is necessary to write code that utilizes COM Interop or that is written based on some platform-specific feature, such as a pointer size. These assemblies would get into the platform-specific section of the Global Assembly Cache (GAC).

Internally, in fact, there are multiple folders maintained for storing these assemblies separately. A look at the \Windows\Assembly folder from the command line (Figure 8) shows the organization of the GAC. Table 1 describes these folders.

Common Language Runtime Changes

The Common Language Runtime (CLR) has undergone internal changes to support the move to 64-bit computing. For the most part, the changes are related to code generation, garbage collection, exception handling, and debugging.

- Code generation. The 64-bit version of the CLR needs to support development of 64-bit native applications. This means that a new Just-In-Time (JIT) Compiler had to be built for each of the new platforms, namely the IA64 and the x64 platforms for generating native code for the specific platforms.
- Garbage collection. A 64-bit processor can support more memory and break the 4-GB memory barrier that existed with 32-bit systems. Hence, the garbage collector has been tuned to support larger memory.
- Exception handling. Exception handling for 64-bit systems has been completely revamped and rewritten, while retaining the end-user experience to be the same.
- Debugging. The debugger is dependent on the code-generation and exception-handling subsystems. Because of the changes to these two subsystems, the debugger also had to undergo changes.

Development Tools

Visual Studio 2005 supports development of 64-bit applications using Visual C++ .NET, Visual C#, and Visual Basic .NET. Visual J# will not support development of

64-bit applications as part of Visual Studio 2005. Figure 9 depicts the various languages supported on Visual Studio 2005 and platforms supported by these managed languages.

The Visual Studio 2005 development environment will ship as a 32-bit application that makes use of the WOW64 system. Most features provided on the 32-bit platform are available on the 64-bit platform. A notable exception is the lack of the Edit and Continue features, which have been reintroduced for the 32-bit versions of C# and VB.NET.

Apart from Visual Studio 2005, the Windows Platform SDK contains a 64-bit compiler toolset, which includes a Visual C++ compiler for developing 64-bit applications.

Precautionary Measures

Before looking at what's involved in developing 64-bit native applications, it is worth asking, "What steps can I take today in my application to ensure that the code will be portable to 64-bit?"

It is possible to take some precautionary measures and the toolset also provides some support. For instance, the Visual C++ compiler supports the /Wp64 switch that detects portability issues that may arise from the source code being compiled.

A similar facility is being built for Visual Studio 2005 by adding support for detecting these compatibility issues at compilation. This is being achieved by adding rules to FxCop, which is now integrated with the Visual Studio 2005 IDE. Although not part of Beta 1, the final release of Visual Studio 2005 will have FxCop-based rules that cover aspects in code that could affect portability of the application.

In the case of the managed languages, these features create issues with portability:

- Interop-related code that involves COM Interop and Platform Invoke: Native 64-bit applications cannot load 32-bit COM DLLs. That is, a 64-bit process cannot transition into 32-bit code and host a 32-bit DLL within the same process. Interop between processor architectures is not possible within the same process. Consequently, when 64-bit applications have to utilize any COM DLLs, a 64-bit version of the COM DLL is required.

  In many cases, however, these COM DLLs may be third-party code to which you do not have access. In such cases, the application needs to be built to target x86 architecture, in which case it runs using the WOW64 subsystem. The other option is to host the 32-bit DLL in a separate 32-bit process and make RPC calls to this host from the 64-bit application.
- Equality comparison of floating-point numbers. It is not guaranteed that the same IL will produce the same results on 32-bit and 64-bit platforms. Hence, it is recommended that the equality comparison not be made directly on floating-point numbers. Floating-point representation on 64-bit computers is based on the IEEE-754 Standard, which allows for differences. The major impact of this is on financial applications and graphics-based applications where precision is important. Algorithms should be designed in such a way that it can handle the skewed values. (For more information, see David Goldberg's paper "What Every Computer Scientist Should Know about Floating-Point Arithmetic," http://docs.sun.com/source/806-3568/ncg_goldberg.html.)
- Explicit control of layout of a structure using the *StructLayout* attribute. The *StructLayoutAttribute* is applied to structures and classes. When it is specified as Explicit, the precise position of each member of an object in unmanaged memory is explicitly controlled. Compared to a 32-bit platform, the packing of structures is different on a 64-bit platform due to the data types used in the structure. Consequently, the use of explicit control of the layout of a structure should be avoided.
- Bitwise operations on numbers. C# provides bitwise operators, which include the bitwise AND, bitwise OR, left-shift, and right-shift operators. Bitwise operations on data types vary from 32-bit to 64-bit computers since the internal representation of the data types vary across the platforms.
- Custom Serialization. The .NET Framework provides two options for serialization — an automatic serialization that can be achieved by using the Serializable attribute, and a custom serialization that can be achieved by getting

Figure 6: *Administrative Tools menu.*

Folder	Description
GAC	Stores the assemblies built for the .NET Framework 1.0/1.1
GAC_32	Stores the 32-bit assemblies built using .NET Framework 2.0.
GAC_64	Stores the 64-bit assemblies built using .NET Framework 2.0.
GAC_MSIL	Stores the portable assemblies; that is, those that have the Processor architecture set as MSIL.

Table 1: *Organization of the GAC.*

Assembly Name	Version	Culture	Public Key Token	Processor Architecture
System.Data.OracleClient	2.0.3600.0		b77a5c561934e089	x86
System.Data.OracleClient	2.0.3600.0		b77a5c561934e089	AMD64
System.Data.SqlXml	2.0.3600.0		b77a5c561934e089	MSIL
System.Deployment	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Design	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.DirectoryServices	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.DirectoryServices.Protocols	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Drawing	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Drawing.Design	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.EnterpriseServices	2.0.3600.0		b03f5f7f11d50a3a	x86
System.EnterpriseServices	2.0.3600.0		b03f5f7f11d50a3a	AMD64
System.Management	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Messaging	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Runtime.Remoting	2.0.3600.0		b77a5c561934e089	MSIL
System.Runtime.Serialization.Formatters.Soap	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Security	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.ServiceProcess	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Transactions	2.0.3600.0		b77a5c561934e089	x86
System.Transactions	2.0.3600.0		b77a5c561934e089	AMD64
System.Web	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Web.Mobile	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Web.RegularExpressions	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Web.Services	2.0.3600.0		b03f5f7f11d50a3a	MSIL
System.Windows.Forms	2.0.3600.0		b77a5c561934e089	MSIL
System.Xml	2.0.3600.0		b77a5c561934e089	MSIL

Figure 7: *Global Assembly Cache.*

the type to implement the *ISerializable* interface. When the underlying serialization mechanism provided by the .NET Framework is utilized, you will not face any problems. However, when custom serialization has been implemented via *ISerializable*, then there's a chance that the results might vary from 32-bit platform to 64-bit platform, depending on the custom approach adopted to achieve the serialization.

```
Visual Studio .NET Whidbey Command Prompt

G:\WINDOWS\assembly>dir
 Volume in drive G is Win2003x64Edition
 Volume Serial Number is 00B7-5487

 Directory of G:\WINDOWS\assembly

10/16/2004  06:39 PM    <DIR>          GAC
10/16/2004  06:39 PM    <DIR>          GAC_32
10/16/2004  06:34 PM    <DIR>          GAC_64
10/16/2004  06:11 PM    <DIR>          GAC_MSIL
10/16/2004  06:34 PM    <DIR>          NativeImages_v2.0.40607_32
10/16/2004  06:29 PM    <DIR>          NativeImages_v2.0.40607_64
10/16/2004  06:11 PM    <DIR>          temp
10/16/2004  06:34 PM    <DIR>          tmp
               0 File(s)              0 bytes
               8 Dir(s)  18,615,918,592 bytes free
```

Figure 8: *The \Windows\Assembly folder from the command line.*

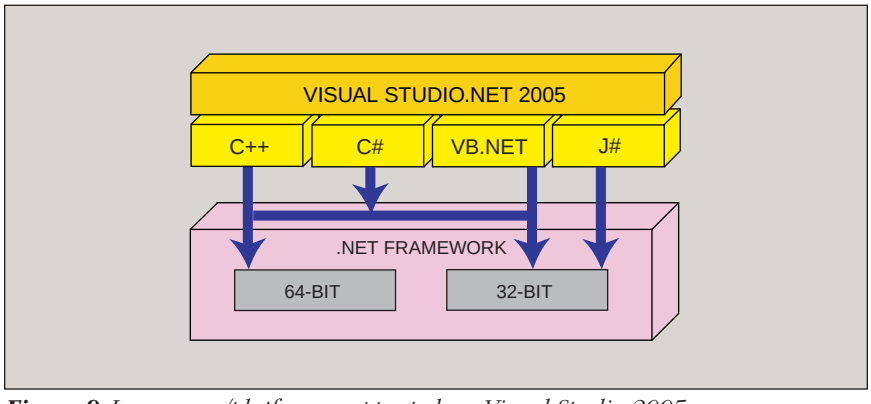


Figure 9: *Languages/platforms supported on Visual Studio 2005.*

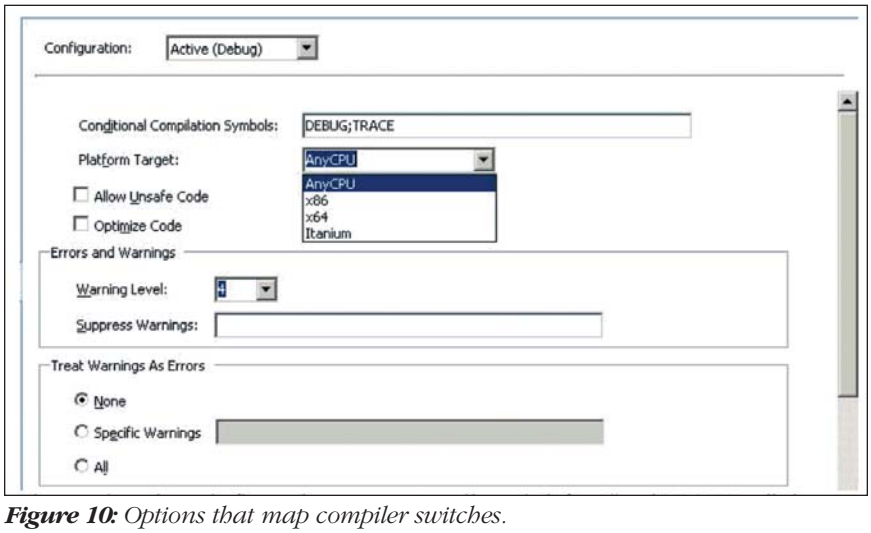


Figure 10: *Options that map compiler switches.*

Platform	Description
AnyCPU	Generates a platform-agnostic assembly. Also known as "portable assembly."
x86	Generates a 32-bit assembly targeting the x86 platform.
x64	Generates a 64-bit assembly targeting the x64 platform.
Itanium	Generates a 64-bit assembly targeting the Itanium platform.

Table 2: *Visual Studio 2005 supports the development of 64-bit applications that target these platforms.*

Of course, there may be times when the application demands use of some of these features. In such scenarios, it is necessary to build and test a 32-bit version and 64-bit version separately.

Development Using Visual Studio 2005

Again, Visual Studio 2005 supports the development of 64-bit applications that target the platforms in Table 2. You get the same user experience with the IDE and build the application as you would any normal application, keeping in mind the previously presented guidelines.

While compiling the application, the target platform can be set on the Property Pages for the Project. The property pages include a Build tab, which lets you specify the platform. In Figure 10, for instance, the options map the compiler switch for C# and VB.NET called "/platform." For

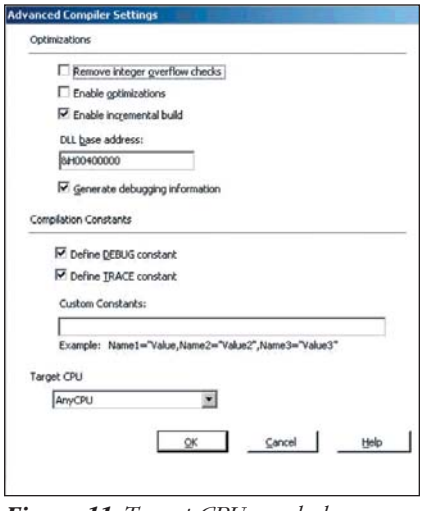


Figure 11: *Target CPU combobox.*

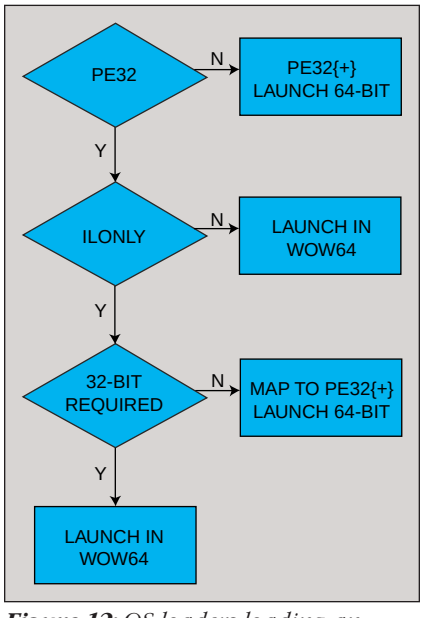


Figure 12: *OS loaders loading an executable.*

VB.NET, the same option is available in the Compile Menu on the Advanced Compiler Settings Dialog Box. The Target CPU combobox in Figure 11 allows for setting the specific CPU that the application requires.

Of course, use of a specific setting depends on the specific scenario:

- *AnyCPU* generates an assembly that is platform agnostic. This is the IDE's default option. An assembly compiled with the *AnyCPU* option can run on x86-, x64-, and Itanium-based systems without problems. The output assembly generated is based on the PE32 (Portable Executable 32-bit) format. (PE32 is the file format defining the structure that all EXEs and DLLs must use.)
- *x86* is used to generate code specific for a 32-bit Intel x86-compatible processor. The generated output assembly is based on the PE32 (Portable Executable) format. The executables generated by setting this option use the WOW64 subsystem.
- *x64* is used to generate code specific to 64-bit native application, which targets the x86-processor architecture. The generated output assembly is based on the PE32 {+} (Portable Executable Plus) format (this is an extension to the existing PE32 file format). These executables will run natively on a 64-bit x64 machine.
- *Itanium* is used to generate code specific to 64-bit native applications, which target the Itanium (IA-64) processor architecture. The generated output assembly is based on the PE32 {+} (Portable Executable Plus) format. These executables will run natively on a 64-bit Itanium machine.

Another important aspect to consider is that the application being developed may need to work on all platforms; consequently, it may not be possible to build a platform-agnostic application. In such cases, the development strategy would be to make use of preprocessor directives such as *#define* and *#if*, along with conditional compilation constants. The code that is specific to a particular target platform is wrapped by preprocessor directives along with conditional compilation constants for the particular platform; the compilation is performed specifying the particular conditional compilation constant.

The recommended conditional compilation constants are:

- *_AMD64_* for code that is specific to the AMD64 platform.
- *_IA64_* for code that is specific to the IA64 platform.
- *_WIN64-* for code that is specific to either of the 64-bit platforms.

Under the Hood: Loading a .NET Executable

Switch settings specified are embedded in the PE32 or the PE32 {+} executable generated. The PE32 {+} format is an extension to the PE32 format and has information regarding the machine type.

"With the PE32, the CLR Header contains additional flags, such as *ILOnly* and *32BitRequired*"

With the PE32, the CLR Header contains additional flags, such as *ILOnly* and *32BitRequired*. The *ILOnly* flag is set when an assembly is built with the platform set to *AnyCpu*. The *32BitRequired* flag is set when the assembly is compiled with the platform set as x86. When the platform is set to x64 or Itanium, a PE32 {+} executable is created with information regarding machine type embedded in the output file.

The operating-system loader loads an executable based on these settings. The control flow is used by the OS Loader in loading an executable; see Figure 12.

When the executable is found to be a PE32 {+}, then the EXE is launched as a 64-bit process. If not, then the *ILOnly* flag is verified. If this flag is not set, then the executable is determined to be a 32-bit executable and launched using the WOW64 subsystem.

When the *ILOnly* flag is set, a further check is made to see if the *32bitRequired* flag is set. When it is set, then the executable is launched in the WOW64 subsystem; otherwise, it is remapped as PE32+ and launched as a 64-bit application.

So Is It Faster?

A question that most people often ask with 64-bit computing is "Are 64-bit applications faster compared to 32-bit applications?" This is a common myth surrounding 64-bit technology. The answer to this question is "maybe." The reason for this answer is that the performance of applications depends on many factors, and it is not possible to make a statement claiming that 64-bit applications are faster. Computing in 64-bit technology enables newer software designs, which can exploit the larger memory that 64-bit processors support. An application that has been designed to take advantage of this larger memory will be able to outperform a similar 32-bit application.

64-Bit Momentum in the Industry

Many vendors who have products for Windows have started releasing products for the 64-bit version of Windows:

- AMD has released a performance analyzer for Windows called "AMD Code Analyst." See http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_869_3604,00.html.
- InstallShield 10.5 supports installation of 64-bit applications. See http://www.installshield.com/downloads/installshield/aag.pdf.
- Compuware has released its DevPartner Studio in its 64-bit form entitled "DevPartner64." See http://www.compuware.com/products/devpartner/64.htm.
- The Java 2 Platform Standard Edition 5.0 (J2SE) for the AMD64 platform is currently available as Release Candidate. See http://javashoplm.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=jdk-1.5.0-rc-windows-amd64-JPR&SiteId=JSC&TransactionId=noreg.
- Hardware manufacturers are releasing 64-bit native drivers for their products. A complete listing can be found at http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_10454,00.html.
- Games have always utilized the latest and greatest hardware; for instance: Unreal Tournament (http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_10220_9486%5E9621~75301,00.html), Far Cry (http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_10543,00.html), and Shadow Ops: Red Mercury (http://www.amd.com/us-en/Processors/ComputingSolutions/0,,30_288_11054_11705,00.html). (The 64-bit enhanced version of Shadow Ops is showcased at http://www.atari.com/shadowops/us/amd.html.)

DDJ

Integer 64-Bit Optimizations

Exploiting the power of 64-bit platforms

ANATOLIY KUZNETSOV

Addressing and 64-bit operations are useful in applications that deal with large amounts of data, such as scientific and engineering applications, large databases, and the like. There are a number of CPUs and operating systems that natively support 64-bit computing. Probably the biggest advantage they provide is a huge available address space, in which applications can allocate more than 4GB of memory, easily maintain large files, and more. But to fully utilize the power of 64-bit CPUs, applications need to exploit the wider machine word. In this article, I focus on performance optimization techniques that take advantage of that power in this way.

64-Bit Safety

Unfortunately, much of today's software doesn't take advantage of 64-bit microprocessors and often can't even be compiled and operated in 64-bit mode. Consequently, software runs in 32-bit compatibility mode—a clear waste of silicon. Moreover, there are a number of common C coding "malpractices" when coding for 32-bit systems with a hypothetical 64-bit CPU in mind:

- Reliance on the fact that the size of *pointer* is equal to the size of *int*. For 64-bit systems, *sizeof(void*) == 8* and *sizeof(int)* usually remains 4. Ignoring this can result in an incorrect assignment and crash.
- Reliance on a particular byteorder in the machine word.
- Using type *long* and presuming that it always has the same size as *int*. Direct assignment of this type causes value truncation and leads to a rare and difficult-to-detect problem.
- Alignment of stack variables. In some cases, stack variables can have addresses not aligned on 8-byte boundaries. If you typecast these variables to 64-bit variables, you can get into trouble on some systems. But if you place a 64-bit variable (*long* or *double*) on the stack, it is guaranteed to be aligned. Heap allocated memory is aligned, too.
- Different alignment rules in structures and classes. For 64-bit architectures, structure members are often aligned on 64-bit boundaries. This leads to problems in sharing binary data through IPC, network, or disk. Packing data structures to save resources can cause problems if alignment is not taken into consideration.
- Pointer arithmetic. When a 64-bit pointer is incremented as a 32-bit pointer, and vice versa. The 64-bit pointer is incremented by 8 bytes and the 32-bit pointer by 4 bytes.
- In the absence of function prototypes, the return value is considered to be *int*, which can cause value truncation.

Parallel Programming: Getting the Most From Each Cycle

The key to high-performance 64-bit C programming is wide integer and FPU registers. CPU registers are at the top of the food chain—the most expensive type of computer memory there is. In 64-bit CPUs, registers are 8-bytes wide, although a corresponding 128- or 256-bits wide memory bus is also common.

Figure 1 illustrates typical operation on a 32-bit system. The CPU crunches data coming from memory 4 bytes at a time. Figure 2 shows that a 64-bit system having wide registers can process 8 bytes at a time.

Listing One performs a bit XOR operation on a block of memory, representing an integer-based bitset. You can optimize this code for 64-bit mode. Listing Two, for instance, relies on the *long long* C type, which is not supported by some compilers. As you can see, I did not change the total size of the bit set block, although it now takes twice fewer operations to recombine vectors. Listing Two reduces the loop overhead and equivalent to the loop unrolling with coefficient 2. The disadvantage of this code, of course, is its pure 64-bitness. Being compiled on a 32-bit system gives a wrong result because of the different *long* size.

You can make further modifications, as in Listing Three, which uses wide registers to do the job on 32-bit and 64-bit CPUs. When typecasting like this, remember pointer alignment. If you blindly typecast *int* pointers to 64-bit *long* pointers, the address might not be 8-bytes aligned. On some architectures, this causes a bus error and crash; on others, it leads to performance penalties. Listing Three is not safe because it is possible that the 32-bit *int* variable placed on the stack will be 4-bytes aligned and the program will crash. Heap allocation (*malloc*) is a guarantee against this occurring.

> "The key to high-performance 64-bit C programming is wide integer and FPU registers"

Bit Counting

One of the most important operations in bit set arithmetic is counting the number of 1-bits in bit strings. The default method splits each integer into four characters and looks up a table containing precalculated bit counts. This linear approach can be improved by using 16-bit-wide tables, but at the cost of a much larger table. Moreover, larger tables will likely introduce some additional memory fetch operations, interfere with a CPU cache, and won't deliver a significant performance boost.

As an alternative, I present code inspired by "Exploiting 64-Bit Parallelism" by Ron Gutman (*DDJ*, September 2000). Listing

Anatoliy is currently working on projects with the National Center for Biotechnology Information and National Institutes of Health. He can be contacted at anatoliy_kuznetsov@yahoo.com.

Four does not use lookup tables, but computes the two *int*s in parallel.

Bit String Lexicographical Comparison

Another application for 64-bit optimization is lexicographical comparison of bitsets. The straightforward implementation takes two words out of the bit sequence and performs bit-over-bit shifting with comparison. This is an iterative algorithm with *O(N/2)* complexity. *N* here is the total number of bits. Listing Five illustrates iterative comparison of two words. This algorithm cannot be significantly improved by 64-bit parallelization. However, Listing Six, an alternative numerical algorithm with complexity proportional to half the number of machine words (not bits), has good 64-bit potential.

The Challenge

The $64,000 question here is whether 64-bit is worth the trouble. Contemporary 32-bit CPUs are superscalar, speculative execution machines that often provide several execution blocks that can execute several commands in parallel and out-of-order, without intervention from programmers. The truth is that 64-bit processors exhibit the same properties and can run code in parallel — but only in 64 bits. Plus, some architectures, such as Intel Itanium, specifically emphasize parallel programming and concentrate efforts on explicit optimization on the compiler level. Making code 64-bit ready and optimized is a necessity in this case.

Another objection is that performance is often limited not by the raw MHz-based CPU performance, but by CPU-memory bandwidth, which is bus limited; our algorithms are not going to show the top performance, anyway. This is a fact of life and hardware designers know it. We all see implementation of high-performance dual-channel memory controllers and steady hikes in the memory speed. This effort certainly makes bus bottlenecks less critical, and optimized 64-bit algorithms are going to be better prepared for modern hardware.

Algorithmic Optimization, Binary Distances

One candidate for 64-bit optimization is the computing of binary distances between bit strings. Binary distances are used in data mining and AI applications doing clustering and finding similarities between objects, which are described by binary descriptors (bit strings). The optimization hotspot here is a distance algorithm, which can be repeated for every pair of objects in the system.

The most-known distance metric is the Hamming distance, which is a minimum number of bits that must be changed to convert one bit string into another. In other words, you combine bit strings using bitwise XOR and compute the number of bits ON in the result.

The starting point for the analysis is code like Listing Seven. The obvious optimization here is to get rid of the temporary bitset and compute both XOR and population count in parallel. The creation of temporaries is a "favorite in-house sport" of C++ compilers and wastes performance on reallocations and memory copying; see Listing Eight.

This optimization immediately achieves several goals: reduction of memory traffic, better register reuse, and, of course, 64-bit parallelism (see Figure 3). The essential goal here is to improve the balance between CPU operations and memory loads. The objective has been achieved by combining the algorithms in Listings Three and Four.

This optimization technique can be further extended on any distance metric that can be described in terms of logical operations and bit counting. What's interesting is that the effect of optimization of more complex metrics like the Tversky Index, Tanamoto, Dice, Cosine function, and others, is more pronounced.

To understand why this is happening, consider the Tversky Index:

```
TI = BITCOUNT(A & B) /
  [a*(BITCOUNT(A-B) +
   b*BITCOUNT(B-A) + BITCOUNT(A & B)]
```

The formula includes three operations: *BITCOUNT_AND(A, B), BITCOUNT_SUB(A, B)* and *BITCOUNT_SUB(B, A)*. All three can be combined into one pipeline; see Figure 4. This technique improves data locality and better reuses CPU caches. It also means fewer CPU stalls and better performance; see Listing Nine.

Is There Life After 64-Bits?

Many of the algorithms I've described can be coded using vector-based instructions, single instruction, multiple data (SIMD). CPUs that are SIMD-enabled include special, extended (64- or 128-bits) registers and execution units capable of loading several machine words and performing operations on all of them in parallel. The most popular SIMD engines are SSE by Intel, 3DNow! by AMD, and AltiVec by Motorola, Apple, and IBM. SIMD registers are different from general-purpose registers; they do not let you execute flow-control operations such as *IF*. This makes SIMD programming rather difficult. Needless to say, portability of SIMD-based code is limited. However, a parallel 64-bit optimized algorithm conceptually can be easily converted to a 128-bit SIMD-based algorithm. For instance, in Listing Ten, an XOR algorithm is implemented using the SSE2 instruction set; I used compiler intrinsics compatible with the Intel C++ compiler.

For More Information

Ron Gutman. "Exploiting 64-Bit Parallelism." *DDJ*, September 2000.

Brian T. Luke. "Clustering Binary Objects" (http://fconnyx.ncifcrf.gov/~lukeb/binclus.html).

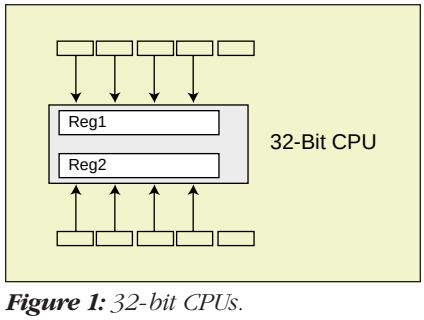


Figure 1: *32-bit CPUs.*

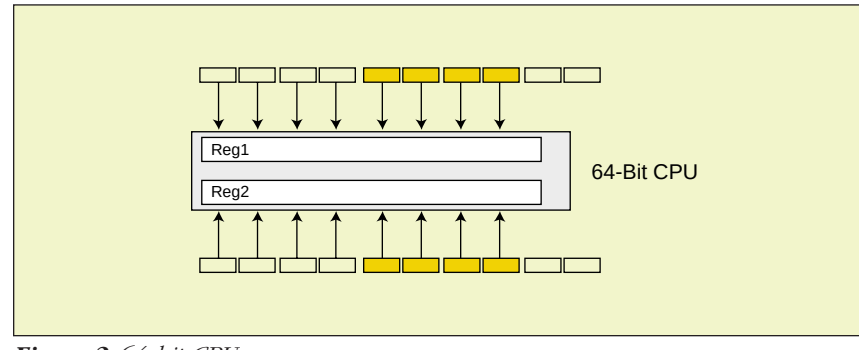


Figure 2: *64-bit CPUs.*

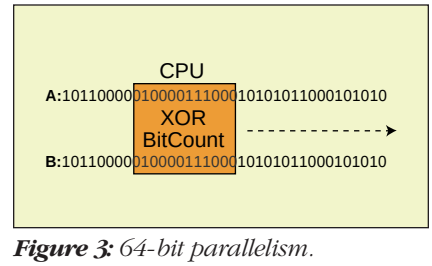


Figure 3: *64-bit parallelism.*

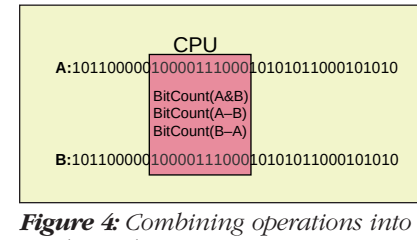


Figure 4: *Combining operations into a single pipeline.*

Ian Witten, Alistair Moffat, and Timothy Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999. ISBN 1558605703.

Wi-Fen Lin and Steven K. Reinhardt. "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design." *Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*.

Intel Corp. "Intel Pentium 4 and Intel Xeon Processor Optimization."

Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley Professional, 2002. ISBN 0201914654.

DDJ

Listing One

```
{
    int   a1[2048];
    int   a2[2048];
    int   a3[2048];

    for (int i = 0; i < 2048; ++i)
    {
         a3[i] = a1[i] ^ a2[i];
    }
}
```

Listing Two

```
{
    long long  a1[1024];
    long long  a2[1024];
    long long  a3[1024];

    for (int i = 0; i < 1024; ++i)
    {
         a3[i] = a1[i] ^ a2[i];
    }
}
```

Listing Three

```
{
    int   a1[2048];
    int   a2[2048];
    int   a3[2048];

    long long* pa1 = (long long*) a1;
    long long* pa2 = (long long*) a2;
    long long* pa3 = (long long*) a3;

    for (int i = 0; i < sizeof(a1) / sizeof(long long); ++i)
    {
         pa3[i] = pa1[i] ^ pa2[i];
    }
}
```

Listing Four

```
int popcount(long long b)
{
     b = (b & 0x5555555555555555LU) + (b >> 1 & 0x5555555555555555LU);
     b = (b & 0x3333333333333333LU) + (b >> 2 & 0x3333333333333333LU);
     b = b + (b >> 4) & 0x0F0F0F0F0F0F0F0FLU;
     b = b + (b >> 8);
     b = b + (b >> 16);
     b = b + (b >> 32) & 0x0000007F;

     return (int) b;
}
```

Listing Five

```
int bitcmp(int w1, int w2)
{
    while (w1 != w2)
    {
        int res = (w1 & 1) - (w2 & 1);
        if (res != 0)
            return res;
        w1 >>= 1;
        w2 >>= 1;
    }
    return 0;
}
```

Listing Six

```
int compare_bit_string(int a1[2048], int a2[2048])
{
    long long* pa1 = (long long*) a1;
    long long* pa2 = (long long*) a2;

    for (int i = 0; i < sizeof(a1) / sizeof(long long); ++i)
    {
         long long w1, w2, diff;
         w1 = a1[i];
         w2 = a2[i];
         diff = w1 ^ w2;
         if (diff)
         {
             return (w1 & diff & -diff) ? 1 : -1;
         }
    }
    return 0;
}
```

Listing Seven

```
#include <bitset>
using namespace std;

const unsigned BSIZE = 1000;
typedef  bitset<BSIZE>  bset;

unsigned int humming_distance(const bset& set1, const bset& set2)
{
    bset xor_result =  set1 ^ set2;
    return xor_result.count();
}
```

Listing Eight

```
{
    unsigned int hamming;
    int   a1[2048];
    int   a2[2048];
    long long* pa1;
    long long* pa2;

    pa1 = (long long*) a1; pa2 = (long long*) a2;
    hamming = 0;

    for (int i = 0; i < sizeof(a1) / sizeof(long long); ++i)
    {
         long long b;
         b = pa1[i] ^ pa2[i];

        b = (b & 0x5555555555555555LU) + (b >> 1 & 0x5555555555555555LU);
        b = (b & 0x3333333333333333LU) + (b >> 2 & 0x3333333333333333LU);
        b = b + (b >> 4) & 0x0F0F0F0F0F0F0F0FLU;
        b = b + (b >> 8);
        b = b + (b >> 16);
        b = b + (b >> 32) & 0x0000007F;

        hamming += b;
    }
}
```

Listing Nine

```
{
    double ti;
    int   a1[2048];
    int   a2[2048];
    long long* pa1;
    long long* pa2;

    pa1 = (long long*) a1; pa2 = (long long*) a2;
    ti = 0;

    for (int i = 0; i < sizeof(a1) / sizeof(long long); ++i)
    {
         long long b1, b2, b3;
         b1 = pa1[i] & pa2[i];
         b2 = pa1[i] & ~pa2[i];
         b3 = pa2[i] & ~pa1[i];

         b1 = popcount(b1);
         b2 = popcount(b2);
         b3 = popcount(b3);

    ti += double(b1) / double(0.4 * b2 + 0.5 * b3 + b1);

    }
}
```

Listing Ten

```
void bit_xor(unsigned* dst, const unsigned* src, unsigned block_size)
{
      const __m128i* wrd_ptr = (__m128i*)src;
      const __m128i* wrd_end = (__m128i*)(src + block_size);
      __m128i* dst_ptr = (__m128i*)dst;

      do
      {
           __m128i xmm1 = _mm_load_si128(wrd_ptr);
           __m128i xmm2 = _mm_load_si128(dst_ptr);

           __m128i xmm1 = _mm_xor_si128(xmm1, xmm2);
           __mm_store_si128(dst_ptr, xmm1);
           ++dst_ptr;
           ++wrd_ptr;

      } while (wrd_ptr < wrd_end);
}
```

DDJ

High-Performance Math Libraries

Who says you can't get performance and accuracy for free?

MICK PONT

The AMD Core Math Library (ACML) is a freely available toolset that provides core math functionality for Advanced Micro Devices' AMD64 64-bit processor (http://www.amd.com/amd64/). Developed by AMD and the Numerical Algorithms Group (http://www.nag.com/), the highly optimized ACML is supported on both Linux and Windows and incorporates BLAS, LAPACK, and FFT routines for use in mathematical, engineering, scientific, and financial applications.

In this article, I examine how you can use high-performance math libraries to speed-up application code, and present tricks used to achieve excellent performance, while still maintaining accuracy. Although I concentrate on ACML, the benefits discussed apply just as much to libraries from other hardware and software vendors.

BLAS

The ACML contains the full range of Basic Linear Algebra Subprograms (BLAS) to deal with basic matrix and vector operations. Level 1 BLAS deal with vector operations, level 2 with matrix/vector operations, and level 3 BLAS with matrix/matrix operations. Since a surprising amount of mathematics and statistics rely at some level on these operations, NAG in the 1980s became part of an international team that designed a standard set of interfaces for these operations. The result was the netlib suite of BLAS reference source code (http://www.netlib.org/blas/).

Although you can compile the netlib code, that is probably not the best way to get optimum performance. Many key BLAS routines — double-precision generalized matrix-matrix multiply (DGEMM), for instance — can benefit massively from tuning to a specific hardware platform, as is the case with the ACML.

For instance, Figure 1 illustrates the difference in performance between using the DGEMM code from netlib compiled with full optimization and the ACML code tuned for the AMD64 — both running on a single-processor 2000MHz AMD Athlon64 machine. (The netlib DGEMM was compiled using the GNU Fortran compiler g77 with -O3 optimization level.) Performance is measured in megaflops — millions of double-precision floating-point operations per second. The theoretical peak speed of the processor used for this graph is 4000 megaflops.

Speedups such as this were not gained by using purely high-level languages like Fortran. In fact, for the ACML version of DGEMM, the performance was gained by a heavy dose of assembly language using blocked algorithms designed to take advantage of the processor's cache memory. Fortran wrapper code was then used to set up blocking and handle "cleanup" cases.

The ACML assembly kernels use (and for best performance you or your compiler will want to use) Streaming SIMD Extension (SSE) instructions. Single Instruction Multiple Data (SIMD) lets the processor work on several floating-point numbers, packed into a long 128-bit register, at the same time.

In Figure 2, the 128-bit registers *xmm1* and *xmm2* each contain four packed 32-bit floating-point numbers. Multiplying *xmm1* by *xmm2* returns the four products, again in packed format. This operation can be performed significantly faster than four separate 32-bit products. (It's worth noting that SSE instructions don't just appear on 64-bit processors. Newer AMD and Intel 32-bit chips also have them, so the aforementioned comments don't apply just to the 64-bit world.)

> "The LAPACK routines gain their speed by proper choice of blocking factor and calls of BLAS assembly kernels"

In general, programming in assembly language is not recommended because the frustration and general maintenance overheads tend to outweigh performance gains. However, in this case, significant performance gains can be achieved in a truly core routine, which can be used by a great deal of other code. The good news is that hardware vendors take on all the pain, so that we high-level developers don't have to — so long as we remember to take advantage of it!

Mick is a senior technical consultant for the Numerical Algorithms Group. He can be contacted at mick@nag.co.uk.

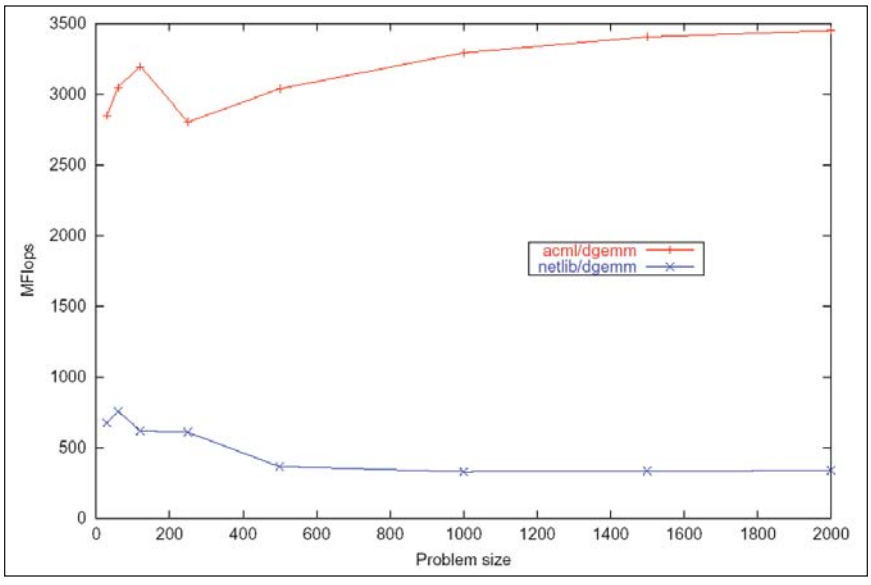


Figure 1: *Timing DGEMM (2.0GHz AMD Athlon64 processor).*

LAPACK

The Linear Algebra Package (LAPACK) is a standard set of routines for solving simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, singular value problems, and the like. LAPACK 3.0 (http://www.netlib.org/lapack/) is available across almost all of the vendor math libraries. LAPACK is designed using blocked algorithms, in which large problems are broken down into smaller blocks wherever possible, to take advantage of level 3 BLAS operations. LAPACK routines cover higher level linear algebra tasks, such as factorizing a matrix or solving a set of linear equations. You can tune many LAPACK routines by choosing a block size that fits in well with the machine architecture. A good choice of block size can lead to many times better performance than a poor choice.

xmm1	a1	a2	a3	a4
xmm2	b1	b2	b3	b4
xmm1*xmm2	a1*b1	a2*b2	a3*b3	a4*b4

Figure 2: *The 128-bit registers* xmm1 *and* xmm2 *each contain four packed 32-bit floating-point numbers.*

Within the ACML, the LAPACK routines gain their speed by proper choice of blocking factor and calls of BLAS assembly kernels, rather than by using their own dedicated assembly. Many key ACML LAPACK routines, such as the matrix factorization routine DGETRF, have also been redesigned internally to take best advantage of the hierarchically cached memory of modern systems like the AMD64. (This matrix factorization is the most important mathematical ingredient of solving a set of simultaneous linear equations.)

Although they maintain identical user interfaces to standard LAPACK, ACML routines have been improved by identifying two kinds of bottleneck:

- Code that must be executed serially, even on a parallel machine.
- Memory access bottlenecks. Sometimes if computations are performed in the natural order, they are required to use data not held in cache, which is slow to operate on.

By rewriting the algorithm used, bottlenecks can often either be removed altogether (perhaps by doing some work earlier than the intermediate results are actually required) or hidden (by postponing work until such time that more can be done with memory-cached data, thus reducing access times).

In addition, ACML LAPACK routines make use of multithreading to achieve extra performance on multiprocessor-shared memory systems. ACML uses OpenMP (http://www.openmp.org/) as a portable way for application programmers to control such shared memory parallelism. (For more information on OpenMP, see "Faster Image Processing with OpenMP," by Henry A. Gabb and Bill Magro; *DDJ*, March 2004.)

Figure 3 shows the performance gain on a multiprocessor system using the LU factorization DGETRF compared to the same code from netlib. The machine used to generate the timing results was a 64-bit 1800MHz four-processor AMD Opteron system. All Fortran code was compiled with the Portland Group (http://www.pgroup.com/) Fortran compiler pgf77, using high-level optimization flags. (Although Opteron is a 64-bit processor, it also runs 32-bit code natively.)

The version of DGETRF compiled from netlib code was built using the vanilla Fortran BLAS code that comes with netlib LAPACK, whereas the ACML version of DGETRF takes advantage of highly tuned ACML BLAS assembly kernels. No additional tuning of netlib DGETRF was performed (knowledgeable users might improve the performance of DGETRF by changing blocking sizes supplied by LAPACK routine ILAENV). The netlib DGETRF is not parallelized, so running on a one-processor or a four-processor machine makes no difference.

In Figure 3, notice that:

- On a single processor, ACML runs over three times as fast as the vanilla netlib Fortran code.

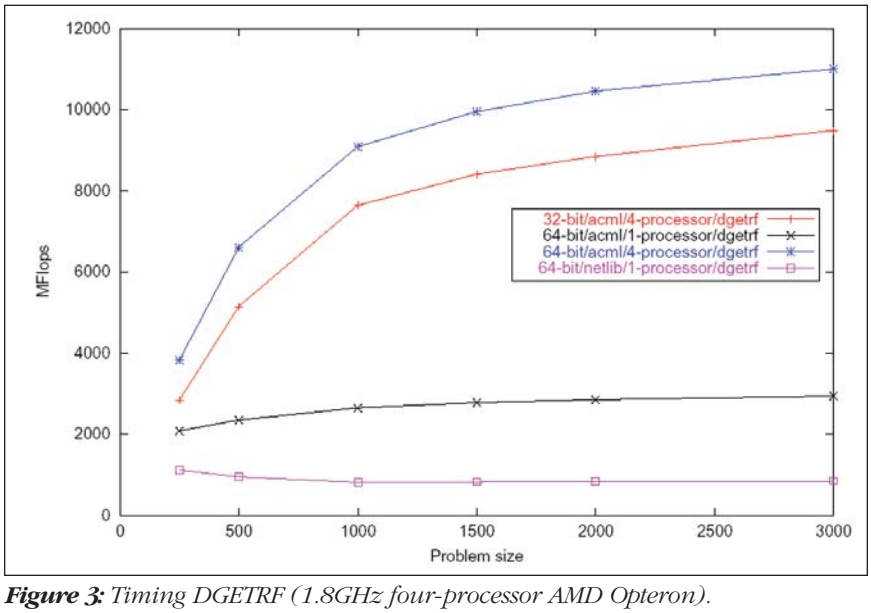


Figure 3: *Timing DGETRF (1.8GHz four-processor AMD Opteron).*

- ACML scales well. Using four processors, DGETRF performs almost four times as fast as using one processor.
- 64-bit code shows a significant advantage. Running both 64-bit and 32-bit versions of DGETRF on four processors, the 64-bit DGETRF runs about 15 percent faster than the 32-bit DGETRF.

In fact, performance gains of the order of 15 percent for 64-bit code over 32-bit code can be seen in a variety of applications. This includes when running on multiple processors or just a single one, despite the fact that AMD64s run 32-bit code at least as fast as 32-bit AMD chips of the same clock speed. This is a very good reason to upgrade to 64-bit code if you can!

Fast Fourier Transforms

Fast Fourier Transforms (FFTs) are the third common suite of routines in vendor math libraries. FFTs can be used in analysis of unsteady signal measurements — the frequency spectrum computed by the FFT tells you something about the frequency content of the signal. For example, an FFT can help you remove periodic background noise from signals without damaging the part of the signal (such as a music recording) that you are interested in.

Unfortunately, FFT routines do not have a common interface, so extra work is necessary when moving between various vendor math libraries. However, given the high-performance gains possible with FFTs, that exercise should be well worth the effort. This is another area where vendors tend to aggressively tune their code. As with the BLAS, ACML makes extensive use of assembly kernels to achieve high performance.

Because of the way FFT algorithms work, much of the performance that can be got from an FFT implementation depends on the size of the FFT to be performed. For any FFT suite, a data sequence with a length that is an exact power of 2 will likely be transformed faster than a sequence of a slightly different size — sometimes many times faster. This is so much the case that some FFT implementations only work on such sequences, and force you to adapt your data to the FFT routine. With ACML, however, FFTs work on any problem size, though it is always best to have a size that is a product of small prime numbers if possible. That is because, in ACML, the code that deals with these small prime factors has been aggressively tuned with assembly code to take advantage of the 64-bit chip, ensuring that data streams into cache memory in the right order to maximize performance.

ACML comes with FFT routines to handle single- and multidimensional data. The multidimensional routines also benefit from the use of OpenMP for good scalability on SMP machines.

Conclusion

When developing code, it is important to build on the work of others and not try to build everything from scratch. Of course, we do it all the time. We do not work in machine code, we program in higher level languages or use packages to insulate us from the boring and mundane!

Sometimes though, we forget to extend those principles of reusing others' tried and tested work as far as possible to gain maximum benefit.

With all this in mind, if in your work you already make use of routines that feature in ACML, why not try linking to ACML? It won't cost you anything, and you just might be pleasantly surprised at the speedup you see in your application. And while this article has concentrated on 64-bit AMD processors, the same concepts apply to 64-bit chips from Intel and others, and high-performance math libraries should be available for most of them.

Finally, if you're keen to work in assembly language, a good resource is the *AMD64 Software Optimization Guide* (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF).

DDJ

Programming with Contracts in C++

Explicitly stating and testing design requirements

CHRISTOPHER DIGGINS

Programming with Contracts (PwC) is a method of developing software using contracts to explicitly state and test design requirements. The contract is used to define the obligations and benefits of program elements such as subroutines and classes. In this article, I explain contracts and present a technique for defining contracts in separate classes safely hidden away from implementation details.

Design by Contract (DbC) is often confused with PwC. Conceived by Bertrand Meyer, DbC is a formal software-design methodology, whereas PwC is the programming technique of using a contractual representation in code to verify that preconditions, postconditions, and invariants are satisfied. The differences can be equated to the relationship between object-oriented programming and object-oriented design. (For more information, http://www.artima.com/intv/contracts.html.)

Contracts

Contracts are made up of three major elements, referred to as "clauses": preconditions, postconditions, and class invariants. Preconditions and postconditions are clauses that are evaluated at the beginning and end of specific routines, respectively. From a design standpoint, a precondition represents the obligations on the context invoking the routine. An example can be found using the C file reading function *fscanf*:

```
int fscanf(FILE* stream,const char * format
[ , argument , ...]);
```

The function *fscanf* operates with the precondition that stream must be a pointer to an open file. There is also another precondition that there must be the same number of type specifiers in the format string as the number of arguments passed. Postconditions, however, represent both the obligations and benefits of a given subroutine. In the case of *fscanf*, the implicit postconditions are:

- The return value is *EOF* if an error occurs; otherwise, it is the number of items successfully read not including any ignored fields.
- The FILE* argument is updated.

From a theoretical standpoint a pre/postcondition should be evaluated at compile time when appropriate. This can be done using *BOOST_STATIC_ASSERT* found in the Boost C++ library (http://boost/static_assert.hpp). Pre/postconditions conceptually are also language independent. In fact, some pre/postconditions are difficult and even impossible to express in a formal language. In these cases, the pre/postcondition is best expressed as a natural language comment.

A class invariant is a property of each instance of a class that is required to evaluate to True before and after every external call to a public function. One way to think of a class invariant is as a clause that is ANDed with the pre- and postcondition of each public method. Checking class invariants in C++ is beyond the scope of this article, but the same effect can be achieved more verbosely with pre/postconditions.

Peppering Code with Assertions: Good, but Good Enough?

Most modern C++ programmers have adopted the good habit of (usually) checking their preconditions and postconditions in their code by placing assertions throughout their routines. This is fine for simple single-expression assertions, as the code generated can be completely removed by turning off assertion compilation.

> "Contracts are made up of three major elements"

Simply using assertions is somewhat unsatisfying as the set of preconditions and postconditions is buried deep in the code and is hard to extract. Automated tools can extract preconditions and postconditions, assuming they follow a consistent naming convention, but for a human perusing the code, the contracts of nontrivial functions and classes are not easily parsed.

The other drawback of assertions is that they only apply to code — if you have nontrivial clauses, then this approach starts to spill into noncontract code. Consider this example:

```
template<typename T>
class Stack {
  void Push(T x) {
    int nOldCount = Count()
    // implementation here
    assert(Count() == nOldCount + 1);
  }
  ...
}
```

Christopher is a freelance computer programmer and developer of Heron, a modern, general-purpose, open-source language inspired by C++, Pascal, and Java. He can be contacted at http://www.heron-language.com/.

Notice that the code for allocating space and initializing *nOldCount* will possibly remain in your executable, regardless of whether assertions are turned off.

Perhaps in this case, a compiler might optimize it away, but for nontrivial examples, it is virtually impossible for optimizers to remove all unreachable code. In this case, the simplest thing to do would be to wrap the *nOldCount* declaration in a *#ifdef/#endif* pair.

```
template<typename T>
class Stack {
  void Push(T x) {
    #ifdef CONTRACT_CHECKING_ON
    int nOldCount = Count()
    #endif // CONTRACT_CHECKING_ON
    // implementation here
    assert(Count() == nOldCount + 1);
  }
  ...
}
```

This may appear verbose, but the advantage that this code is effectively embedded with a test case that disappears entirely during release builds should be quite clear.

Writing code in this manner assures that the code is continually tested every time the code is executed. Retrofitting comparable test cases after code is already written is harder and almost invariably not as effective.

However, I have good news if you find this overly complex— there is a better way.

Using Contract Classes

The contract clearly needs to be separated from the implementation details. To separate it, you can define it in its own class. This makes the contract more easily parsed, and makes it reusable. The way I implement PwC in Heron— and which applies equally well to C++— is to define the contract as a template class that inherits from its primary type parameter, which you then use to wrap an implementation class. Consider this implementation class:

```
struct SortableIntArray_impl {
  bool SortRange(int i, int j);
  int GetAt(int i);
  void SetAt(int i, int x);
    };
```

You can define the contract of the three public functions in code as in Listing One. Despite being conceptually simple, this contract required a significant amount of code to express. If this had been embedded directly in the implementation itself, it would become messy and confusing.

The contract can now be applied easily to the object conditionally as in Listing Two. The same effect can also be achieved using metatemplate programming techniques involving type selectors such as the STLSoft library compile-time function *stlsoft::select_first_type* or the Boost library compile-time function *mpl::if_c*.

Conclusion

Using assertions does allow the implementation of PwC, but using contract classes is better. Contract classes express and validate significantly complex contracts without obfuscating the implementation. Contract classes make design intentions explicit and improve the readability of code. Contract classes can also often be reused for classes with different implementations but with similar interfaces.

Acknowledgment

Thanks to Matthew Wilson for reviewing the article and pointing out that writing contracts that involve system conditions is a bad idea.

DDJ

Listing One

```
template <typename T>
struct SortableIntArray_contract : public T {
  bool SortRange(int i, int j) {
    // preconditions
    assert(i >= 0);
    assert(i < size());
    assert(j >= i);
    assert(j < size());
    // implementation
    T::SortRange(i, j);
    // postconditions
    // in essence, is this array sorted
    for (int n=i; n < j; n++) {
      assert(get_at(n) <= get_at(n=1));
    }
  }
  int GetAt(int i) {
    // preconditions
    assert(i >= 0);
    assert(i <= size());
    return T::GetAt(i);
  }
  void SetAt(int  i, int x);
    // preconditions
    assert(i >= 0);
    assert(i <= size());
    T::SetAt(i, x);
    // postcondition
    assert(T::GetAt(i) == x);
  }
};
```

Listing Two

```
#ifdef CONTRACT_CHECKING_ON
   typedef SortableIntArray_contract<SortableIntArray_impl> SortableIntArray;
#else
   typedef SortableIntArray_impl SortableIntArray;
#endif
```

DDJ

Making a Scene with Java3D

Creating realistic graphics in 3D

MICHAEL PILONE

Go ahead, admit it—you've wasted way too many hours playing the latest first-person-shooter game to hit the shelves. Don't worry, you aren't alone. The realistic environments, quick action, and competitive play make the games irresistible. This is due in part to a wonderful use of three-dimensional (3D) graphics (but mostly due to the love of fragging your friends, which is an article for another time). However, 3D graphics are not just limited to the gaming world. Many industries now rely heavily on 3D graphics for data visualization, building and component design, medical research, virtual tours, and so on. Advertising, especially TV commercials, is also making heavy use of 3D animation and special effects.

Over the past few years computer graphics hardware has made incredible strides with faster CPUs and Graphical Processing Units (GPU) at constantly decreasing prices. On the software front, OpenGL, officially introduced in 1992, has become the standard API for high-speed 3D graphics programming. As a procedural interface developed in C, OpenGL is incredibly powerful, robust, and stable. However, learning OpenGL can be time consuming and, as with all procedural languages, code maintenance and extension can be difficult on large projects. Enter the 3D scenegraph.

Michael is a software engineer and researcher for the Department of Defense at the Naval Research Laboratory in Washington, D.C. Michael also founded and functions as CTO of Zizworks Inc. (http://www.zizworks.com/), a web-application and custom software development company. He can be contacted at mpilone@botch.com.

A scenegraph provides an object-oriented and logical representation of a 3D scene. Scenegraphs are implemented in many languages and many scenegraphs are simply abstraction layers above the OpenGL rendering library. This abstraction is the foundation of Java3D, a scenegraph API designed and developed by Sun Microsystems for the Java platform. Java3D offers a large 3D API and scenegraph structure to help you write maintainable, scalable 3D applications quickly. In this article, I examine scenegraphs in detail and present an example of a Java3D application.

Java3D is a free library for the Java platform (http://java.sun.com/products/java-media/3D/). At its most basic level, Java3D provides a scenegraph and 3D rendering context for creating graphics applications. However, that description doesn't give nearly enough information. Some of the top Java3D features include:

- Multithreaded scenegraph rendering and stimulus processing.
- Fog, lighting, level of detail (LOD), and sound support.
- Geometry and texture processing and serialization.
- Low-level API abstraction (supporting both OpenGL and Microsoft's DirectX).
- Vector math operations and full Java Foundation Class (JFC) library support.

As with any Java application, Java3D applications are cross platform to Solaris, Windows, Linux, and Apple OS X. (Java3D is available on OS X in a limited fashion through Apple's developer program. More information can be found at Apple's web site, not Sun's.) Also, Java3D applications are web deployable using Sun's Webstart technology (https://j3d-webstart.dev.java.net/). Backed by OpenGL or DirectX, Java3D boasts impressive rendering speed by allowing these highly optimized libraries to do the rendering work and making use of the native graphics hardware and software drivers. All of these features packed into a free API create a powerful tool.

Understanding the Scenegraph

To understand scenegraphs, it is important to know how a scenegraph differs from the procedural or "pipeline" model of 3D programming. Using a library like OpenGL, the application procedurally defines all of the triangles, textures, and other graphics primitives to draw one after another for each drawing cycle. All of this information is pushed into the graphics pipeline and to the graphics card. Unfortunately, the procedural model can force the application to either send a lot of wasted data into the pipeline, because it is not visible in the current view. Likewise, the application may be forced to perform complex math operations to cull (or remove) unseen information before sending it to the card. The procedural model also requires that the application maintain the state of the scene so that it can be redrawn whenever required.

> "A scenegraph provides an object-oriented and logical representation of a 3D scene"

To solve some of the pipeline rendering limitations and to make developing graphics applications friendlier, the scenegraph paradigm was introduced. A scenegraph is a hierarchical graph of a scene or virtual world. The scenegraph is composed of nodes, which represent mathematical transformations, lighting, shapes, and views. On each rendering cycle, a renderer walks this graph from the top to the bottom, performing many optimizations such as culling nodes that cannot be seen, collapsing nodes that can be combined, and compiling nodes for future rendering. By performing these optimizations, the renderer can limit the number of primitives that get sent to the underlying rendering library and hardware. A scenegraph

also provides a logical representation of where objects are in the virtual world and lets you interact with nodes directly in a more object-oriented fashion. On the downside, a scenegraph can introduce larger memory requirements to an application, as well as the time and CPU cycles required to continually traverse the graph.

Java3D is an implementation of the scenegraph paradigm. In Java3D, the scenegraph is encapsulated in a *VirtualUniverse*, which contains a directed, acyclic graph (DAG) of *Nodes*, either leaves or groups, such as: *BranchGroups, TransformGroups, Shape3Ds, Lights,* and *ViewingPlatforms*. Each node has a specific purpose in the tree and because the graph is a DAG, there is only one possible path to any node in the scene. *Nodes* may in turn contain *NodeComponents* to represent items such as appearance and geometry. *NodeComponents* are not considered part of the graph; therefore, they may be shared between nodes. Figure 1, a simple Java3D scenegraph, contains one branch that defines the viewer of the scene, and one branch from the root that defines the content of the scene. More branches can be added, but this simple scenegraph presents all of the basic concepts. The Java3D renderer is continually rendering the scene, starting at the root node, and traversing down the scene with each *TransformGroup* applied as it is encountered and each node rendered. Again, the Java3D renderer is able to perform simple view-frustum culling at this stage of the rendering process, far before the data is pushed to the video card.

Getting Your 3D Feet Wet

The first step in creating a Java3D application is to create the virtual universe, which contains the entire scene. Sun provides the utility class *SimpleUniverse* to make this process straightforward, although a custom universe can be constructed with the *VirtualUniverse, View, PhysicalBody,* and *PhysicalEnvironment* classes, but they are beyond the scope of this article. Listing One creates the *SimpleUniverse* object with the default settings of the utility class. The *SimpleUniverse* requires a canvas for the renderer to draw into. Similar to the Advanced Windowing Toolkit (AWT) *Canvas* class, the Java3D *Canvas3D* class provides a rendering context that can be added to any AWT or Swing container and it behaves as a heavyweight component. The *SimpleUniverse* class automatically builds the view side of the scenegraph for you with a standard layout that will work for simple applications.

Once the universe has been created, it is time for the fun stuff— creating the scene content. The content needs to attach to a root group node. The basic group node in Java3D is the *BranchGroup,* which can have any number of children and serves as merely a branching point in the tree. Listing Two is the root group along with another type of group node, a *TransformGroup*— a node containing a 3D transform matrix that applies to the rendering pipeline as the renderer encounters the group in the tree traversal. As with all 3D graphics programming, the placement and orientation of objects in Java3D is determined by the application of matrices to objects. Two transforms are defined in Listing Two— one to scale the content to fit in the canvas, another to rotate any of its children (anything attached to this group) by 35 degrees around the x-axis. In this application, the rotation is simply done to show that the object in the scene truly is 3D. In more complex applications, you make use of many transformations to move the viewer, position objects, simulate animation, and so on. Once the transform group is created, it must be added to the root group as a child. This addition of children is how the graph is built to represent the scene.

At this point there is nothing in the scene for the renderer to actually draw, such as a shape. Java3D defines a *Shape3D* class as the root for almost all renderable objects in a scene. A *Shape3D* object contains an *Appearance* and one or more *Geometry* components that may be shared between shapes. The *Appearance* component of a shape defines elements such as color, material, transparency, and drawing attributes. The *Geometry* component of a shape defines the actual 3D points and lines that are drawn by the underlying graphics library, if the renderer determines that the given shape should be rendered. Once again, to simplify the task of creating shapes, appearances, and geometry, Sun has provided a few utility classes that neatly and efficiently define some common shapes: *Box, Cone, Sphere,* and *ColorCube*. To add a *ColorCube* shape to the scene, add:

```
ColorCube cube = new ColorCube(.5);
objRoll.addChild(cube);
```

The cube is created with an edge length of 0.5 units and it is added as a child of the rotation transform that was added to the scene previously. Now, add the root group to the universe to assemble the final scene:

```
universe.addBranchGraph(rootBG);
```

Once the scene is assembled and the canvas is displayed, the Java3D renderer immediately begins rendering in a separate thread. The result is the rendered 3D cube, like that in Figure 2.

Behaviors & Interactions

Although an impressive result for such little code, a static 3D scene is not very useful. At some point you are going to require elements such as user interaction, animation, effects, and movement. To accomplish these tasks, Java3D provides a behavior system that works alongside the renderer to provide hooks that allow the application to be notified of events in the scene. Behavior objects are scenegraph nodes like many of the nodes you have already seen and can be added to the scenegraph to perform many functions in response to a large range of stimuli. A small sample of possible stimuli includes:

- A desired number of frames elapsing.
- Collision of 3D objects.
- Mouse and keyboard events.
- A desired amount of time elapsing.

These stimuli are defined by subclasses of the *WakeupCondition* class, which has many other useful extensions.

Built on top of the behavior system are *Interpolator* classes that can be used to smoothly move or transform an object in the scene, which is useful for view transitions, morphing, or animation. Listing Three presents modifications to the scenegraph that was constructed above to add

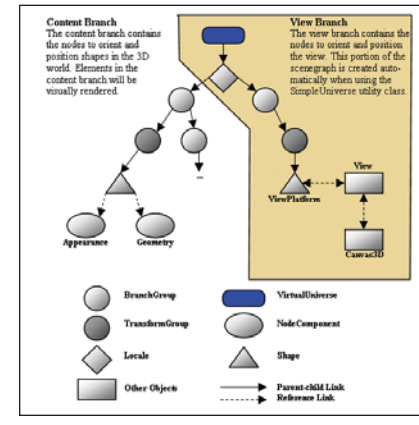


Figure 1: *Simple Java3D scenegraph.*

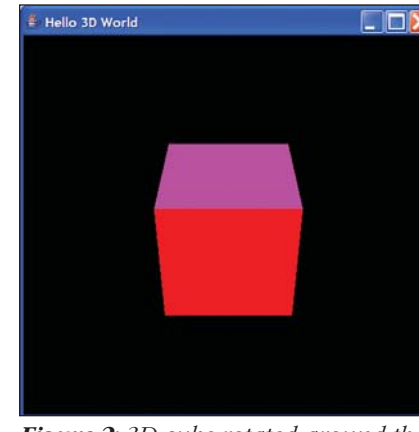


Figure 2: *3D cube rotated around the x-axis.*

another transform group, which will rotate its children about the y-axis. When combined with a *RotationInterpolator*, the cube appears as a spinning cube in the scene, updated each frame by the interpolator (which, remember, is a *Behavior*). The effect of the spinning is visible in Figure 3. (A Java3D application demonstrating scenegraph creation using transforms, a predefined shape, and an interpolator behavior is available electronically; see "Resource Center," page 5.)

Tips & Tricks

So far, I have only presented some of the core Java3D concepts. Once you get going with Java3D programming, you may find some tips and tricks useful. These tips may also help you to avoid some of the traps that many new Java3D developers fall into.

Java3D supports complex canvas configurations including multiple canvases and views for the same scene. This means it is possible to render a single scenegraph (a single universe) in many windows simultaneously, either from the same viewpoint in the scene or from many different view points. Combine multiple canvases with a canvas configured for off-screen rendering and it is possible to create dynamic 3D snapshots for web pages. Be aware that the *Canvas3D* class does have peculiarities. A common problem many developers have with the *Canvas3D* class is that it is a heavyweight Swing component. This can cause some problems in applications that mix heavyweight and lightweight components, such as *JPopupMenus*. Be sure to read up on the limitations of mixing these components at http://java.sun.com/products/jfc/tsc/articles/mixing/index.html.

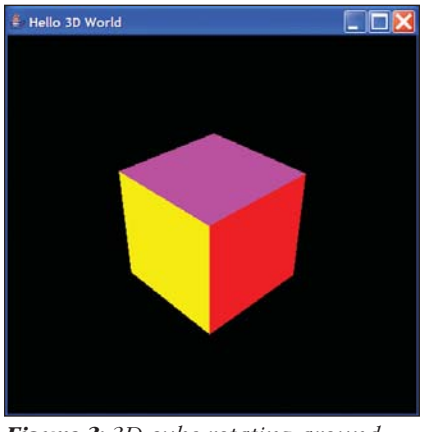


Figure 3: *3D cube rotating around the y-axis.*

Figure 4: *Collection of available Java3D applications, from top to bottom: Builder RF Visualization Tool, Cassos, CazaPool3D, FlyingGuns, Law and Order II.*

3D applications have a tendency to require a lot of memory due to the geometry definitions and textures required. The *-X* command-line options of your JVM may let you increase the heap size for the application. In Sun's JVM, the *-Xmx* and *-Xms* flags perform wonders. Before undertaking any large Java3D application, be sure to think about the overall design of the application. Because the renderer is continuously rendering in a separate thread, any scenegraph modifications may become immediately visible. If this is not the desired effect (say, you require atomic updates), consider using a *Behavior* object. Sun, with the help of community developers (http://www.j3d.org/), has composed a document describing items for performance tuning: http://www.j3d.org/tutorials/quick_fix/perf_guide_1_3.html.

Conclusion

Java3D is a powerful library; however, it is not the only 3D API for Java. Numerous lower level APIs are available to provide direct access to the OpenGL rendering library such as JOGL (https://jogl.dev.java.net/) and LWJGL (http://java-game-lib.sourceforge.net/). Also, competitor scenegraph implementations exist, with the most popular and stable being Xith3D (http://xith.org/). Currently, it appears that the Sun Java3D developers are recognizing and encouraging Xith3D as a high-speed, gaming-oriented scenegraph, while Java3D takes a more user-friendly, thread-safe, visualization approach. Luckily, the Xith3D developers have kept the API generally similar to Java3D, so knowledge in one can be easily transferred. Each 3D API has advantages and disadvantages, so review them all before starting a major project.

It seems that Sun is once again behind Java3D, allocating resources toward its development and integration into widely publicized projects, such as its own Project Looking Glass (https://lg3d.dev.java.net/). Many Java3D developers have also released free applications and games that give a glimpse of what is possible with Java3D. Commercial products using Java3D can be found, such as the "Law and Order" game from Legacy Interactive (http://www

.lawandordergame.com/). There also appears to be a number of scientific research applications using Java3D that never get public attention; however, discussion commonly occurs on the mailing lists. Figure 4 is a collection of screenshots from a few applications written in Java3D. Sun has recently released Java3D as an open-source project, inviting developers to contribute patches and new frameworks to the API for consideration, which has brought new life and energy to the project.

Java3D is a large API, containing more than 100 core classes and many more utility classes. Learning the entire package is a hefty undertaking. Luckily there are many resources available for more information, such as the Java3D mailing lists and forums, a great tutorial at the Java3D web site, and a few books. Combining Java and Java3D with some of the other powerful Java APIs such as Java Advanced Imaging (JAI), Java2D, and the Java Media Framework (JMF), it is possible to create robust cross-platform applications.

DDJ

Listing One

```
// The canvas needs some information about the graphics environment. This
// information could be custom built if desired, but a utility method
// exists to make this easier.
GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();

// Create the canvas which will serve as the rendering surface. The
// canvas is a component like any AWT component, therefore it can
// be added to a JFrame to be displayed.
Canvas3D canvas = new Canvas3D(gc);

// A SimpleUniverse is a utility class that wraps some of the VirtualUniverse
// configuration options and sets up a basic universe that is useful for
// simple demonstrations. The universe serves as the root of the scenegraph.
SimpleUniverse universe = new SimpleUniverse(canvas);

// Get the viewing platform from the universe and set a nominal
// transform. This will move the viewer slightly back from the
// center so you can see the nodes in the scene.
universe.getViewingPlatform().setNominalViewingTransform();
```

Listing Two

```
// Root group of scene graph. Everything is created as a child of this group.
BranchGroup rootBg = new BranchGroup();

// Create a simple transform to scale scene down so it fits in the view.
Transform3D scaleTrans = new Transform3D();
scaleTrans.setScale(0.6);
TransformGroup objScale = new TransformGroup(scaleTrans);
rootBg.addChild(objScale);

// Create a simple transform to rotate around the x
// axis to show that the cube really is 3 dimensional.
Transform3D rollTrans = new Transform3D();
rollTrans.rotX(Math.toRadians(35));
TransformGroup objRoll = new TransformGroup(rollTrans);
objScale.addChild(objRoll);
```

Listing Three

```
// Create a transform to rotate the shape using an interpolator. Once the
// transform group is added to the scene, Java3D won't allow modifications
// unless you tell it that you want that capability, therefore you set the
// ALLOW_TRANSFORM_WRITE.
TransformGroup objRotate = new TransformGroup();
objRotate.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objRoll.addChild(objRotate);

// Create an interpolator behavior object that will rotate the cube
// by modifying the rotation transform at runtime.
Transform3D yAxis = new Transform3D();
Alpha rotationAlpha = new Alpha(-1, Alpha.INCREASING_ENABLE,
                                0, 0, 8000, 0, 0, 0, 0, 0);
// Setup the scheduling bounds of the behavior so it runs indefinitely.
Bounds bounds = new BoundingSphere(new Point3d(0, 0, 0), 100.0);

// Create the interpolator that will rotate the given transform
// around the y axis as the alpha value changes.
RotationInterpolator rotator =
    new RotationInterpolator(rotationAlpha, objRotate, yAxis,
                             0.0f, (float) Math.PI*2.0f);
rotator.setSchedulingBounds(bounds);
objRotate.addChild(rotator);
```

DDJ

A Sound File Editor for Netbeans

A full-featured Java IDE built on top of the Netbeans Platform

RICH UNGER

I am on the development team for V-Builder, an IDE for building VoiceXML-based speech applications. Among other features, we had to provide V-Builder with the ability to edit several file types you don't normally associate with IDEs—call flows, VoiceXML files, linguistic grammars, recorded prompts, and the like. To implement these features, we turned to Netbeans, an open-source framework for building Java client applications (http://www.netbeans.org/), mainly because much of the necessary functionality for IDEs is already implemented in Netbeans. It has a windowing system, JavaHelp integration, source control, syntax coloring and completion, generic XML editing, and lots of other goodies. Consequently, we decided to implement our application as a set of Netbeans plug-in modules.

For instance, one of our modules is a prompt editor. In the context of a speech application, a prompt is a file containing information about questions the application may ask users. The information includes:

- A transcript.
- Instructions for the voice talent recording the prompt (optional).
- The recorded .wav file (optional).

The first two items are stored in a Java properties file with the extension "prompt." An important design consideration is that, because applications are designed before the voice talent records the prompts, it is possible to have the prompt file without the .wav file. So, our Netbeans module must be able to recognize prompt files by themselves, and associate them with sibling .wav files in a seamless fashion.

The Anatomy of a Module

A Netbeans module is a jar file with an enhanced manifest. To turn any existing jar file into a Netbeans module, you add two lines to the jar's manifest:

```
OpenIDE-Module: org.netbeans.modules
                                .mymodule/1
OpenIDE-Module-Specification-Version: 1.0
```

"A Netbeans module is a jar file with an enhanced manifest"

This is sufficient for Netbeans to install, recognize the code in that jar file, and assign a class loader to create instances of the classes defined there. Of course, I needed to do more than this to get my code to integrate into the Netbeans UI. The prompt editor's manifest (Listing One) declares a few more fields that affect the runtime behavior of the module. All

Rich is a software engineer at Nuance Communications, and a member of the Netbeans Governance Board. He can be reached at richunger@netbeans.org.

Netbeans-specific entries in the manifest begin with "OpenIDE-Module."

The *OpenIDE-Module-IDE-Dependencies* field indicates that the module requires Version 4.41 of the core framework (which corresponds to Netbeans 4.0). The *OpenIDE-Module-Module-Dependencies* field declares dependencies on classes from other modules. For example, I use the *org.openide.io* module to print information to the output window. The *OpenIDE-Module-Localizing-Bundle* field points to a properties file containing manifest entries that may be translated into different languages, such as the title, description, and category of this module (Listing Two). The *Class-Path* entry adds jar files to the module's class loader. In this case, the only declared jar file contains a JavaHelp help set, which can be loaded from menu items or Netbeans' context-sensitive help system.

The *OpenIDE-Module-Layer* field indicates the location of the layer file, an XML document that declares how the UI components integrate with the rest of the framework. Netbeans treats its UI like a filesystem. For example, the Menus folder contains a folder for each top-level menu. These can, in turn, contain folders (submenus) or files (menu items). The prompt editor's layer file (Listing Three) is an XML representation of part of this filesystem. At startup, Netbeans takes all the modules' layers and merges them together to create the complete picture.

The final two lines of my manifest declare a *DataLoader* class, which describes a file type. Netbeans maintains a loader pool, which scans files in a given directory, grouping the files into logical chunks, or just determining what type of data each represents. Most *DataLoader*s extend *UniFileLoader*, and recognize individual files based on their extensions or MIME types. The *PromptLoader* (Listing Four) is a *MultiFileLoader*, because I want a .wav file and prompt file to appear as a single item in the file explorer. The *findPrimaryFile()* method knows how to pair the files together. The *createMultiObject()* method knows how to create a *DataObject* from a prompt file.

A *DataObject* represents a particular instance of a file or group of files. A *PromptDataObject* (Listing Five, available electronically; see "Resource Center," page 5) represents a single prompt file and its associated .wav file. The *DataObject* is responsible for encapsulating the relevant data, as well as creating node representations of the *DataObject*, tracking whether the data has been modified since the last save, and maintaining a set of cookies.

Cookies are capabilities (open, save, edit, print, and so on) that are different from actions. A cookie represents the capability to perform an operation. The action represents the UI component for performing the operation. For example, a *PromptNode* always has a *SaveAction* associated with it. The *SaveCookie*, however, is added or removed from the *PromptDataObject* when it is modified or saved. This is because users should not be able to save files that have not been modified. The action remains, even when the cookie is gone. The result is that the action is disabled (grayed-out).

Prompts are edited using a *PromptEditor* (Listing Six, available electronically), which is an instance of *TopComponent*. A *TopComponent* is a JComponent that is managed by the Netbeans window manager. Most top components in Netbeans actually subclass *CloneableTopComponent*, which adds the ability to create more than one view of the same *DataObject*.

The *PromptEditor* has a single member variable — an *EditorPanel*. This is the actual implementation of the editor, which could be referenced from a JFrame in a standalone Swing application, or (in my case) from the *PromptEditor*. The important things to override in a *TopComponent* are the open/close behavior and serialization routines. The *open()* hook in *PromptEditor* simply defers to the open behavior in *EditorPanel*, which renders the audio waveform, transcript, and instructions. The *canClose()* hook is the

usual place to check if the underlying data was modified, and save (or ask to save) as necessary.

Serialization is controlled with three hooks. The *readExternal()* and *writeExternal()* hooks are not specific to Netbeans, and should be familiar to anyone writing Java GUI applications. The third, *getPersistenceType()*, controls whether Netbeans should bother to serialize a *TopComponent* on shutdown. There are three possible values: PERSISTENCE_NEVER indicates that this component should not be restored upon restarting Netbeans; PERSISTENCE_ONLY_OPENED indicates that the component should only be restored if it was visible when Netbeans was shut down; PERSISTENCE_ALWAYS indicates that the component should always remember where the window was docked, even if it was closed at the time Netbeans exited.

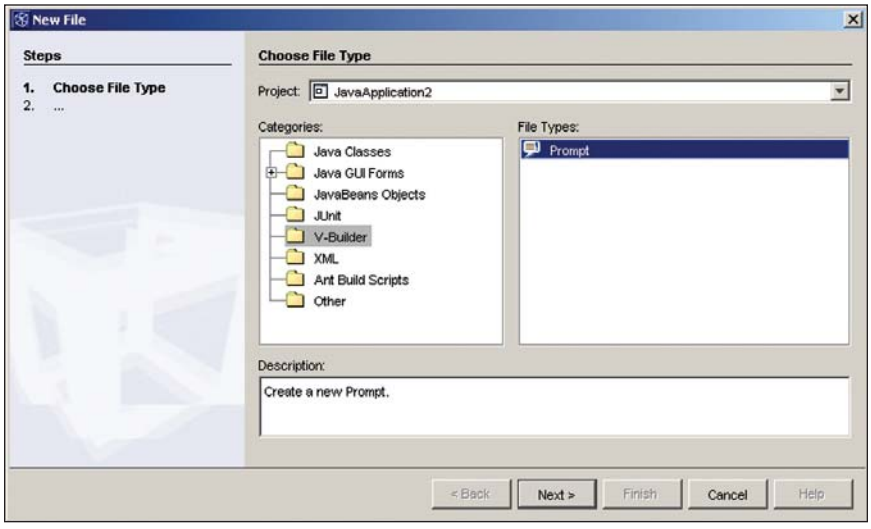


Figure 1: *Selecting File|New File from the menu.*

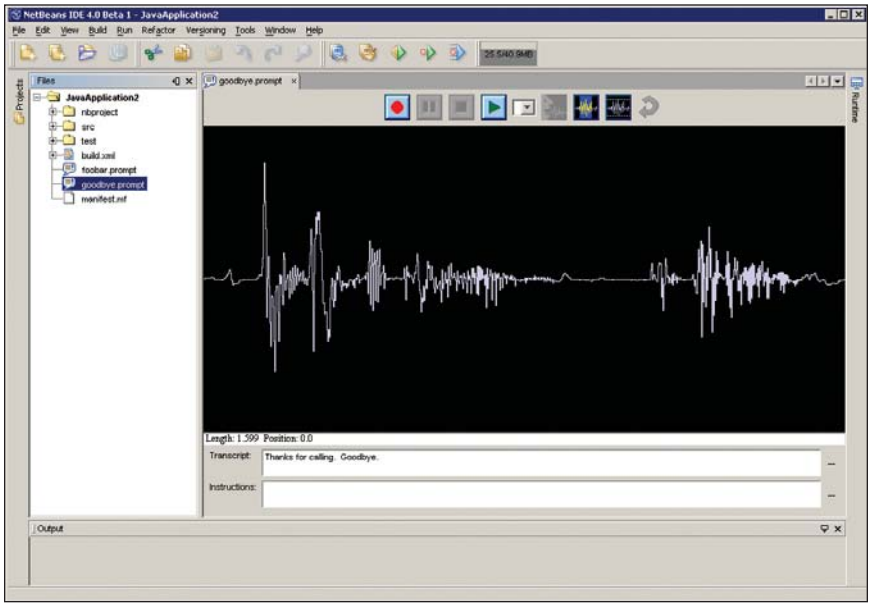


Figure 2: *Double-click to edit.*

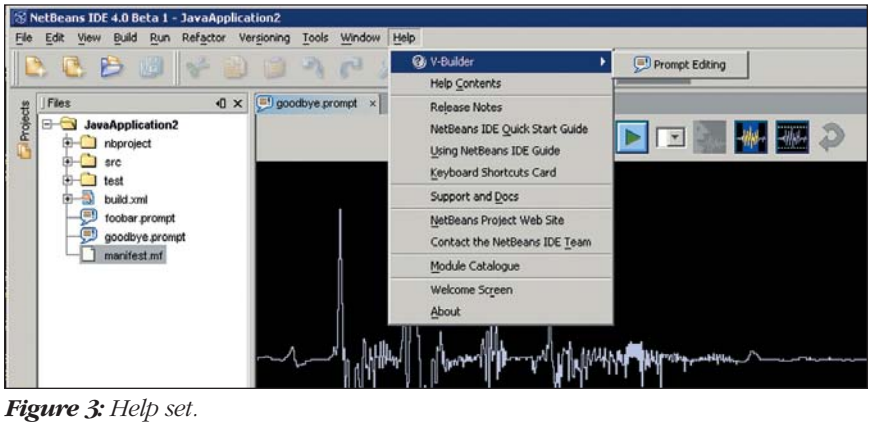


Figure 3: *Help set.*

Testing the Prompt Editor

By creating a jar file from this source code, layer file, and manifest, you have a complete, integrated Netbeans module. The quickest way to test this module is from within Netbeans. The Tools|Options menu item displays a tree view of configuration options. In this tree, under IDE Configuration|System, there is a node called "Modules." Right-click this node and select Add|Module, then select the jar file in the resulting file dialog.

Now you can select File|New File from the menu and see Prompt as one of the templates (Figure 1). Create a prompt this way, and you should be able to edit it by double-clicking it (Figure 2). Also note the help set referred to in the layer file (Figure 3).

To package the prompt editor for distribution, the usual method is to create a Netbeans module package. This is essentially a signed jar with the extension "nbm," which contains the module jar file and any associated resources to be distributed along with the code. The prompt editor includes the JavaHelp help set this way. Netbeans provides an Ant task called "<makenbm>" that packages everything for you.

Conclusion

The prompt module exercises just a few of the many integration points modules can make with the Netbeans framework. Building client applications on top of Netbeans lets you concentrate on writing the functionality necessary to address your core competencies, rather than reinventing the GUI application wheel.

V-Builder is a good example of leveraging the entire Netbeans IDE to create an IDE for building something besides Java applications. However, the Netbeans Platform is also an excellent framework for building applications that are not IDEs.

DDJ

(Listings begin on page 52.)

Listing One

```
Manifest-Version: 1.0
OpenIDE-Module: com.nuance.tools.prompt/1

OpenIDE-Module-Specification-Version: 1.0
OpenIDE-Module-IDE-Dependencies: IDE/1 > 4.41
OpenIDE-Module-Module-Dependencies: org.openide.io
OpenIDE-Module-Localizing-Bundle:
                    com/nuance/tools/prompt/resources/Bundle.properties
OpenIDE-Module-Layer: com/nuance/tools/prompt/resources/mf-layer.xml
Class-Path: docs/com-nuance-tools-prompt-edit.jar

Name: com/nuance/tools/prompt/PromptLoader.class
OpenIDE-Module-Class: Loader
```

Listing Two

```
# Bundle.properties
# moved from the module manifest, so they can be localized

OpenIDE-Module-Name=Prompt Editor
OpenIDE-Module-Short-Description=Prompt Editor
OpenIDE-Module-Long-Description=
          Use this module to record, play, crop and normalize audio files.
OpenIDE-Module-Implementation-Title=V-Builder
OpenIDE-Module-Implementation-Vendor=Nuance
OpenIDE-Module-Display-Category=V-Builder
```

Listing Three

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC '-//NetBeans//DTD Filesystem 1.0//EN'
                            'http://www.netbeans.org/dtds/filesystem-1_0.dtd'>
<filesystem>
  <!-- The "File...New File" templates -->
  <folder name="Templates">
    <folder name="V-Builder">
      <!-- empty.prompt is a file in the same directory as this layer file -->
      <file name="untitled.prompt" url="empty.prompt">
        <attr name="template"
              boolvalue="true"/>
        <attr name="SystemFileSystem.localizingBundle"
              stringvalue="com.nuance.tools.prompt.Bundle" />
        <attr name="SystemFileSystem.icon" urlvalue=
             "nbresloc:/com/nuance/tools/prompt/resources/wavIconSmall.gif"/>
        <attr name="templateWizardURL" urlvalue=
             "nbresloc:/com/nuance/tools/prompt/resources/templatesWav.html"/>
      </file>
    </folder>
  </folder>
  <!-- register the JavaHelp help set -->
  <folder name="Services">
    <folder name="JavaHelp">
        <file name="com-nuance-tools-prompt-edit-helpset.xml">
        <![CDATA[<?xml version="1.0"?>
            <!DOCTYPE helpsetref PUBLIC
            "-//NetBeans//DTD JavaHelp Help Set Reference 1.0//EN"
            "http://www.netbeans.org/dtds/helpsetref-1_0.dtd">
            <helpsetref url="nbdocs:/com-nuance-tools-prompt-edit/
                                          com-nuance-tools-prompt-edit.hs"/>
        ]]>
        </file>
    </folder>
  </folder>
  <!-- make a menu item for the help set -->
  <folder name="Menu">
    <folder name="Help">
      <!-- Put "V-Builder" sub-menu before the "Help Contents" sub-menu -->
      <attr name="V-Builder/HelpShortcuts" boolvalue="true"/>
      <folder name="V-Builder">
        <attr name="SystemFileSystem.icon" urlvalue=
              "nbresloc:/org/netbeans/modules/javahelp/resources/help.gif"/>
        <file name="com-nuance-tools-prompt-edit-help-menu.xml">
          <![CDATA[<?xml version="1.0"?>
          <!DOCTYPE helpctx PUBLIC "-//NetBeans//DTD Help Context 1.0//EN"
          "http://www.netbeans.org/dtds/helpcontext-1_0.dtd">
          <helpctx id="com.nuance.tools.prompt.edit" showmaster="false"/>
          ]]>
          <attr name="SystemFileSystem.localizingBundle"
                stringvalue="com.nuance.tools.prompt.resources.Bundle"/>
          <attr name="SystemFileSystem.icon" urlvalue=
                "nbresloc:/com/nuance/tools/prompt/resources/wavIconSmall.gif"/>
        </file>
      </folder>
    </folder>
  </folder>
</filesystem>
```

Listing Four

```
package com.nuance.tools.prompt;

import java.io.IOException;

import org.openide.actions.*;
import org.openide.filesystems.FileObject;
import org.openide.filesystems.FileUtil;
import org.openide.loaders.DataObjectExistsException;
import org.openide.loaders.ExtensionList;
import org.openide.loaders.FileEntry;
import org.openide.loaders.MultiDataObject;
import org.openide.loaders.MultiFileLoader;
import org.openide.util.NbBundle;
import org.openide.util.actions.SystemAction;

/**  Recognizes .prompt and .wav files as a single DataObject.
 * .prompt files are the primary file objects. @author Rich Unger
 */
public class PromptLoader extends MultiFileLoader {
    public static final String PROP_EXTENSIONS = "extensions"; // NOI18N
    public static final String WAV_EXTENSION = "wav";
    public static final String INFO_FILE_EXTENSION = "prompt";
    private static final long serialVersionUID = -4579746482156153693L;
    public PromptLoader() {
        super("com.nuance.tools.prompt.PromptDataObject");
    }
    protected String defaultDisplayName () {
        return NbBundle.getMessage(PromptLoader.class, "LBL_loaderName");
    }
    protected SystemAction[] defaultActions () {
        return new SystemAction[] {
            SystemAction.get (OpenAction.class),
            SystemAction.get (FileSystemAction.class), null,
            SystemAction.get (CutAction.class),
            SystemAction.get (CopyAction.class),
            SystemAction.get (PasteAction.class), null,
            SystemAction.get (DeleteAction.class),
            SystemAction.get (RenameAction.class), null,
            SystemAction.get (PropertiesAction.class),
        };
    }
    protected MultiDataObject createMultiObject (FileObject primaryFile)
    throws DataObjectExistsException, IOException {
        return new PromptDataObject(primaryFile, this);
    }
    /** For a given file find the primary file. @param fo the file to find
     * the primary file for @return the primary file for this file or null
     * if this file is not recognized by this loader.
     */
    protected FileObject findPrimaryFile(FileObject fo) {
        // never recognize folders.
        if (fo.isFolder()) return null;
        String ext = fo.getExt();
        if (ext.equalsIgnoreCase(WAV_EXTENSION)) {
            FileObject info = FileUtil.findBrother(fo, INFO_FILE_EXTENSION);
            if(info != null) {
                return info;
            }
            else {
                try {
                    info = fo.getParent().createData(
                            fo.getName(), INFO_FILE_EXTENSION);
                    return info;
                } catch(IOException ioe) {
                    // could not create .prompt file,
                    // so cannot recognize .wav file
                    return null;
                }
            }
        }
        if (getExtensions().isRegistered(fo)) {
            return fo;
        }
        return null;
    }
    /** Create the primary file entry. Primary files are the property files
     * (which contain the prompt's * description and recording instructions).
     * @param primaryFile primary file recognized by this loader
     * @return primary entry for that file
     */
    protected MultiDataObject.Entry createPrimaryEntry(
            MultiDataObject obj, FileObject primaryFile) {
        return new FileEntry(obj, primaryFile);
    }
    /** Create a secondary file entry.
     * Secondary files are wav files, which should also be retained (so, not a
     * FileEntry.Numb object)
     * @param secondaryFile secondary file to create entry for
     * @return the entry
     */
    protected MultiDataObject.Entry createSecondaryEntry(
            MultiDataObject obj, FileObject secondaryFile) {
        return new FileEntry(obj, secondaryFile);
    }
    /** @return The list of extensions this loader recognizes. */
    public ExtensionList getExtensions() {
        ExtensionList extensions =(ExtensionList)getProperty(PROP_EXTENSIONS);
        if (extensions == null) {
            extensions = new ExtensionList();
            extensions.addExtension(INFO_FILE_EXTENSION);
            extensions.addExtension(WAV_EXTENSION);
            putProperty(PROP_EXTENSIONS, extensions, false);
        }
        return extensions;
    }
    /** Sets the extension list for this data loader.
     * @param ext new list of extensions.
     */
    public void setExtensions(ExtensionList ext) {
        putProperty(PROP_EXTENSIONS, ext, true);
    }
}
```

DDJ

Resource Management in Python

When portability, robustness, and performance are important

OLIVER SCHOENBORN

A resource is something that is useful and in limited supply. This includes everything from computer memory and disk space, to filehandles/sockets and threads, mutexes, and semaphores. It is therefore important that resources be returned to the system when they are no longer in use. Failure to do so eventually results in poor performance and ultimately in "starvation"—insufficient memory for the next operation, insufficient disk space, inability to create new threads, and the like. More often, however, it leads to nasty bugs, such as lost data and deadlocks.

Python does a good job of making resource management almost trivial. However, there are some important subtleties that can have a serious effect on the portability, robustness, performance, or even correctness of your Python programs. In this article, I discuss those subtleties and some of the modules and techniques you can use to get around them.

In Python, resources are not available directly but are wrapped in higher level Python objects that you instantiate and use in your own Python objects and functions. Resource management consists of the tasks that you and/or the Python interpreter must carry out to ensure that a resource that you have acquired is returned to the system.

There are actually three issues that may not be immediately apparent to you in Python's resource management model:

- NonDeterministic destruction (NDD).
- Circular references.
- Uncaught exceptions.

NDD

Resource management in Python is trivial 90 percent of the time because it is automated: Once you no longer need an object (say after you return from a function), you can just forget about it, and the interpreter does its best to eventually release it.

The "eventually" part of this statement refers to the fact that the Python Language Reference Manual (http://docs.python.org/ref/ref.html) guarantees that objects stay alive as long as they are in use. This means that if you create an object *foo* in a function *A* and from there call another function *B* that creates a global reference to *foo*, *foo* stays alive past the return from function *A*, at least until that global reference is discarded. There is, therefore, no way for you to know when your object will no longer be in use.

The "its best" refers to the fact that the Python Language Reference (PLR) does not prescribe what the interpreter must do with *foo* once there are no other objects left referring to it; for instance, after *foo* has become unreachable or "garbage." Actually, the PLR does not even prescribe that the interpreter must notice at all. Many languages have that as well, Java and C# being two such examples. Therefore, "its best" depends on the interpreter implementation; CPython, Jython, and IronPython are the most well known. This indeterminism is a trap that is difficult for newcomers to Python to discern because of the *del* operator and *__del__* class method. The *del* operator simply removes a variable name from the local or global namespace, it says nothing about object destruction. On the other hand, the *__del__* method is called automatically by the interpreter just before the object is destroyed. But as explained earlier, object destruction is not guaranteed to occur. The call to your object's *__del__* is completely out of your control and may never happen, and using the *del* operator will not help. The only reliable use for the *del* operator is to make sure you (or a user of your module) can't mistakenly use a name that shouldn't be used.

> "Resource management in Python is trivial 90 percent of the time"

This can be even harder to accept for many Pythoneers who only have experience with the CPython implementation: That implementation appears deterministic because in trivial examples such as *a = foo(); del a*, the *foo* is immediately destroyed. Consequently, you may not realize that your code will not work with other interpreter implementations.

This indeterminism is even a trap for experienced programmers who have a background in object-oriented languages that have deterministic destruction (such as C++), where create-release pairs (for instance, the "resource acquisition is initialization," or "RAII" idiom) are used heavily, and to great effect. It is tempting to see *del* as the equivalent of *delete*, and *__del__* as equivalent to a destructor. The two statements taken separately are true, but the difference from C++ is that Python's "delete" does not call the "destructor."

Oliver is a researcher for the National Research Council of Canada doing R&D in simulation systems for engineering applications in virtual reality. He can be contacted at oliver.schoenborn@utoronto.ca.

In fact, *__del__* is not only effectively useless for lifetime management, but should be avoided due to reference cycles.

Circular References

In Python, variable names are references to objects, not objects in and of themselves. A Python interpreter tracks whether an object is in use by how many variable names refer to that object — its "reference count" (which, in the CPython implementation of Python, would be the same as returned by *sys.getrefcount(obj)*). Unfortunately, even if the PLR prescribed that objects *shall* be destroyed, and subjected to this *as soon as* they are no longer referenced by any variable names, you wouldn't be much further ahead because of reference cycles. A reference cycle occurs when an object directly or indirectly refers to itself. For instance, Listing One leads an *A (aa)* to refer to a *B (aa.b)*, which in turn refers to *aa* via *aa.b.a*, and *aa -> aa.b -> aa*, thus creating a reference cycle.

Reference cycles are more common than you might think. For example, they are common in GUI libraries, where information must flow up/down GUI component trees.

While reference cycles are not a problem per se, they do prevent the reference count of all objects in the cycle from going to zero — even those not directly in the chain. So, those objects are never destroyed and associated resources are never released. Hence, reference cycles are a problem only when there are critical resources to release; for example, mutex locks, opened files, and the like, or if cycles get continuously created (as in a loop), which leads to an ever-increasing memory consumption of your program (Listing Two).

Even if you are careful not to create reference cycles, third-party modules that create a cycle that refers to your object, even indirectly, can trap it in the cycle. For instance, the hypothetical object *yourObj* in Figure 1 gets involved in a cycle, *c1->c2->c3->c1*, unbeknownst to you. Its reference count can't go to zero until you break the cycle.

Uncaught Exceptions

Even if you could be sure that reference cycles don't occur in your program, the uncaught exceptions problem remains. An exception in a function causes the named references at all levels of the function call stack to be retained until the Python interpreter exits, in case you need to explore the data for debugging purposes. Yet, objects that are left over from uncaught exceptions are not guaranteed to be destroyed, as stated in the PLR. See Listing Three for a trick that can help clean up resources (but again, the PLR offers no guarantees).

It should be clear by now that you can't rely on *__del__* for resource management; for example, RAII does not work in Python, and the *del* operator is no help in that matter.

Fighting Indeterminism

Listing Four is C++ code that uses the RAII idiom. The C++ code guarantees that the critical mutex lock resource acquired when instantiating a *Lock* is released when the function returns, because *Lock* has a destructor (not shown) that releases it; this is guaranteed by the language standard to be called upon scope exit (that is, function *return*). The equivalent Python code is only equivalent in appearance: *lock* is only a named reference to a *Lock* instance object, so upon scope exit, the reference count of the created *Lock* is decreased, but:

- Python doesn't guarantee that the reference count will be zero, since *do_stuff()* might have increased it.
- Even if *do_stuff()* didn't affect the reference count, the *lock* named reference is not deleted if the scope is being exited due to an exception raised by *do_stuff()*.
- Even if no exception is raised, Python doesn't guarantee that any special function (*__del__*, for instance) will be called.

Until the next release of Python, the only solution is to make judicious use of the *try-finally* clause and manually call the release method (Listing Five). This works

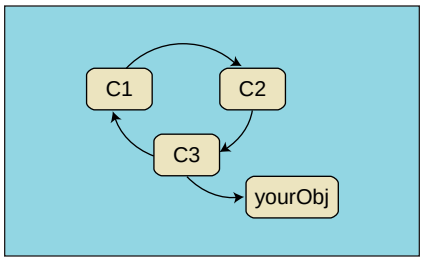


Figure 1: *Cyclic references.*

well, except that the coder of *Lock* must remember to document which method must be called to release resource (not too bad), but the user must read the documentation and notice it (far less likely). In addition, users must remember to use a *try-finally* clause. The latter is actually easy to forget or to get wrong (Listing Six). Also, you can't mix *except* and *finally* clauses. Rather, a *try-except* must wrap a *try-finally*, or vice versa (Listing Seven). This is an unfortunate obfuscation of code that begs for refactoring into a separate function.

Starting with Python 2.4, a new type of expression lets you use the keyword *with*. This lets you write Listing Seven as Listing Eight. This does the right thing as long as the developer of *Lock* has defined an *__exit__* method in *Lock*. Outside of the *with* block, the object that *lock* refers to should not be used. This new syntax cleans things up somewhat but still leaves it up to you to remember that *Lock* requires proper manual resource management; legacy code will not be able to take advantage of this feature (though adding an *__exit__* would be easy to do manually). Also, the PEP (PEP310, http://www.python.org/peps/pep-0310.html) doesn't allow for multiple variables on the *with* line, though that is likely to be a rare requirement.

In cases where a *with* block is still not adequate (for instance, if you have more than one object to guard with the *with* and don't want to nest two *with* clauses), your only options are to:

- Continue with *try-finally* and *try-except*.
- Go with something like detscope.py that provides a means of automating the *try-finally* mechanics (see Listing Nine for a functional prototype that is, however, not multithreadsafe).
- Develop your own technique.

Breaking the Cycle

Since Python 2.1, the standard weakref module supports the concepts of weak reference. A *weakref.ref(yourObject)* is an object that does not affect the reference count of *yourObject*. It has other nice properties, such as being testable for nullness, and letting you specify a callback that gets called when *yourObject* is destroyed; see Listing Ten. A weak reference can be used to break a reference cycle because it tells the interpreter "Don't keep this object alive on my account." Listing Eleven does not create any cycle.

There is a catch: Reference cycles can be hard to find. Since Python 2.1, a garbage collector has been added that detects cycles and frees as many trapped objects as possible. The garbage collector can destroy an object involved in a cycle only if that object does not have a *__del__* method. This is simply because a cycle has no beginning and no end, so there is no way of knowing which object's *__del__* should be called first. Cycles that cannot be destroyed are put in a special list that you can manipulate via the gc module.

The gc module gives access to the garbage collector. The most useful members of gc are:

- *garbage*, a list containing all objects with reference count > *1* but that your code can no longer reach, and which has a *__del__* method.
- *collect()*, forces a cleanup to update *gc.garbage*.
- *get_referrers(obj)*, gets a list of objects that refer to *obj*.
- *get_referents(obj)*, gets a list of objects referred to by *obj*.

Listing Twelve provides code useful for exploring the concept of cycles, and Listing Thirteen is a sample session using it and showing how the cycle is broken. Remember that *gc.garbage* is a standard Python list so it holds "hard" references to your objects: If you manually break a cycle, you must also remove your object from this list for it to be labeled as "unused" (reference count = 0). (See http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/65333/ for a recipe to dump unreachable objects using gc.garbage.)

If you're working with versions of Python prior to 2.1, you can use Cyclops (http://www.python.org/ftp/). This package, not in the standard library, contains the cyclops module, which helps identify reference cycles. Contrary to Python's garbage collector, Cyclops seems to require that you tell it which objects to inspect for cycles.

Once you have identified that a cycle is being created, you must figure out how (where) to break the cycle, or, if that is not possible, how to properly free the critical resources being held some other way. Most often, this will involve wrapping an object with a weakref.ref or weakref.proxy.

Conclusion

If you use Python for its portability, keep both platform and interpreter portability in mind: You must be careful not to rely on your objects being destroyed, ever, so you can't use the RAII idiom. The *del* operator affects only whether you can access (reach) an object, not the object's existence. Therefore, you must make sure you read the documentation of a class to see if a special disposal method must be called when you no longer need the object, and use such disposal methods inside a *try-finally* clause. In Python 2.4, you should be able to use the *with* clause, but that still requires work on your part. Detscope.py and other techniques may be appropriate as well. Circular references may prevent such a disposal from taking place, in which case you must hunt down the cycles manually or using pdb or gc or Cyclops, and fix them using weak references available via the weakref module.

Acknowledgments

Thanks to Holger Dürer, Todd MacCulloch, Francis Moore, and Pierre Rouleau for their helpful reviews of the drafts of this article.

DDJ

Listing One

```
class A:
    def __init__(self):
        self.b = B(self)
    def __del__(self):
        print "goodbye"
class B:
    def __init__(self, a):
        self.a = a
aa = A()
del aa
```

Listing Two

```
while 1:
    aa = A()
```

Listing Three

```
# someScript.py
def run_application():
    ...
def handle_exception():
    ...
try:
    run_application()
except: # catch all
    handle_exception()
    # *attempt* to free as many remaining as possible
    import sys
    sys.exc_clear()
    sys.exc_traceback = None
    sys.last_traceback = None
```

Listing Four

```
# C++ code:
void func() {
    Lock lock;
    do_stuff();
}
# "equivalent" Python code:
def func():
    lock = Lock()
    do_stuff()
```

Listing Five

```
def func():
    lock = Lock()
    try:
        do_stuff()
    finally:
        lock.release()
```

Listing Six

```
# extreme danger:
open('somefile.txt','w').write(contents)
# runtime error in exception handler:
try:
    ff = open('somefile.txt','w')
    ff.write(contents)
finally:
    ff.close() #bang!! ff undefined
# multithreaded:
ff = open('somefile.txt','w')
# if exception raised before getting
# into the try-finally clause: bang!
try:
    ff.write(contents)
finally:
    ff.close()
```

Listing Seven

```
def func():
    lock = Lock()
    # unfortunately not allowed:
    # try:
    #     do_stuff()
    # except MyExcept:
    #     undo_stuff()
    # finally:
    #     lock.release()

    # instead, nesting is necessary:
    try:
        try:
            do_stuff()
        except MyExcept:
            undo_stuff()
    finally:
        lock.release()
```

Listing Eight

```
def func():
    with lock = Lock():
        do_stuff()
```

Listing Nine

```
# detscope.py
"""
Example use: a function, funcWCriticalRes(), creates two critical resources,
of type CritRes1 and CritRes2, and you want those resources to be released
regardless of control flow in function:
  import detscope.py
  def funcWCriticalRes():
      critres1 = CritRes1()
      critres2 = CritRes2()
      use_res(res1, res2)
      if something:
          return # early return
      ...
  funcWCriticalRes = ScopeGuarded(funcWCriticalRes)

  class CritRes1(NeedsFinalization):
      def __init__(self, ...):
          ...
      def _finalize(self):
          ...
  class CritRes2(NeedsFinalization):
      def __init__(self, ...):
          ...
      def _finalize(self):
          ...
"""
import sys
def ScopeGuarded(func):
    return lambda *args, **kwargs: ScopeGuardian(func, *args, **kwargs)
_funcStack = []
class NeedsFinalization:
    def __init__(self):
        print '\n%s: being created' % repr(self)
        self.__finalized = False
        try: _funcStack[-1].append(self)
        except IndexError:
            raise RuntimeError, "Forgot to scope-guard function? "
    def finalizeMaster(self):
        """Derived classes MUST define a self._finalize() method,
        where they do their finalization for scope exit."""
        print '%s: Finalize() being called' % repr(self)
        self._finalize()
        self.__finalized = True
    def __del__(self):
        """This just does some error checking, probably want to remove
        in production in case derived objects involved in cycles."""
        try:
            problem = not self.__finalized
        except AttributeError:
            msg = '%s: NeedsFinalization.__init__ not called for %s' \
                  % (repr(self), self.__class__)
            raise RuntimeError, msg
        if not problem:
            print '%s: Finalized properly' % repr(self)
        else:
            print 'Forgot to scope-guard func?'
def ScopeGuardian(func, *args, **kwargs):
    try:
        scopedObjs = []
        _funcStack.append(scopedObjs)
        func(*args, **kwargs)
    finally:
        _funcStack.pop()
        if scopedObjs != []:
            scopedObjs.reverse() # destroy in reverse order from creation
            for obj in scopedObjs:
                obj.finalizeMaster()
```

Listing Ten

```
import weakref
class Foo:
    def __str__(self):
        return "I'm a Foo and I'm ok"
    def __del__(self):
        print "obj %s: I was a Foo and now I'm dead" % id(self)
def noticeDeath(wr):
    print "weakref %s: weakly ref'd obj has died" % id(wr)
yourObj = Foo()
wr = weakref.ref(yourObj, noticeDeath)
print 'weakref %s -> obj %s: %s' % (id(wr), id(wr()), wr())

del yourObj
assert wr() is None
# output:
#   weakref 17797504 -> obj 17794632: I'm a Foo and I'm ok
#   weakref 17797504: weakly ref'd obj has died
#   obj 17794632: I was a Foo and now I'm dead
```

Listing Eleven

```
import weakref
class A:
    def __init__(self):
        self.b = B(self)
    def __del__(self):
        print "goodbye"
class B:
    def __init__(self, a):
        self.a = weakref.ref(a)
aa = A()
del aa
```

Listing Twelve

```
# testCycle.py
from sys import getrefcount
import gc
class CycleContainer:
    def __init__(self, instName):
        self.instName = instName
        self.cycle = Cycle(self)
        print "Constructed a CycleContainer named '%s'" % instName
    def refs(self):
        """Get number of references to self. The 3 was
        determined experimentally, so method returns
        expected number of references."""
        return getrefcount(self)-3
    def __del__(self):
        """Will prevent CycleContainer instance from being destroyed by gc"""
        print "CycleContainer '%s' being finalized" % self.instName
class Cycle:
    def __init__(self, containerOfSelf):
        self.container = containerOfSelf
def checkgc():
    gc.collect()
    return gc.garbage
```

Listing Thirteen

```
>>> from testCycle import CycleContainer, Cycle, checkgc
>>> aa= CycleContainer('one')
Constructed a CycleContainer named 'one'
>>> aa.refs()
1
>>> aa.cycle = Cycle(aa)
>>> aa.refs()
2
>>> checkgc()
[]
>>> del aa
>>> checkgc()
[<testCycle.CycleContainer instance at 0x00984CB0>]
>>> checkgc()[0].refs()
2
>>> bad = checkgc()[0]
>>> del bad.cycle
>>> bad.refs()
2
>>> checkgc()
[<testCycle.CycleContainer instance at 0x00984CB0>]
>>> del checkgc()[:]
>>> checkgc()
[]
>>> bad.refs()
1
>>> del bad
CycleContainer 'one' being finalized
```

DDJ

The StatiC Compiler & Language

A dual-mode system for sequential and FSM development

PETE GRAY

Take one traditional sequential programming language—C, for instance—and remove the parts you don't like. Design and code a compiler based on this language, and target a popular microcontroller (making sure that the design is flexible enough to support multiple targets). Now, add Finite State Machine programming constructs as an integral part of the language. What you end up with is "StatiC," a dual-mode compiler that provides an easy migration path from the traditional sequential programming model to the inherently multitasking Finite State Machine model. Effectively, you get two languages for the price of one, along with a friendly syntax and small learning curve. Additionally, the use of command-line switches facilitates retargeting through external parsing software.

The StatiC compiler supports both traditional sequential and Finite State Machine (FSM) language methodologies, the feature being controlled via a command-line switch. Dual-methodology support lets you code using an identical syntax (but different language constructs) in either the "generic" sequential mode or the inherently multitasking FSM mode. The sequential language is based on the familiar language constructs of C, Basic, and Pascal, but with a unified and simplified syntax. The FSM language is the same as the sequential language, except that it has additional FSM extensions. In this article, I don't specifically address Finite State Machines, as this topic has been covered in *DDJ* in the past. Rather, I deal with the implementation of the concept as it applies to the StatiC language.

The compiler has been designed from scratch, specifically for the embedded domain, and includes the features required to support both the sequential and FSM modes of operation. In addition, the languages themselves have been enhanced to remove "clutter" (such as ambiguous operators and symbols) found in other languages, as well as incorporating some features more suited for embedded software development.

The StatiC compiler can be hosted under Windows or Linux, and currently targets the Motorola DSP56F80x microcontrollers. These controllers, with their dual Hardvard architecture, JTAG flash capabilities, and a wide variety of interface modules, are particularly well suited for the embedded/robotics domain. A demo version of the compiler is available at http://petegray.newmicros.com/static/ and from *DDJ* (see "Resource Center," page 5).

StatiC compiler operation is controlled via parameters and switches, which are invoked like this:

```
static   sourcename [switches]
```

By convention, StatiC FSM-mode programs have a filetype of .fsm and non-FSM programs have a filetype of .nsm.

Table 1(a) lists the switches that control compiler operation. For example, to produce CodeWarrior-style assembly language for myprog.nsm, enter:

```
static   myprog.nsm    -a568cw
```

Output is placed in the file clist.asm. Table 1(b) lists additional switches for compiler development and debugging.

If the assembler output is unspecified (that is, you don't use the *-a* switch), the compiler generates descriptive, nonoptimized assembler, making it possible for an external program to parse the output and produce assembler for a completely

> "Dual-methodology support lets you code using an identical syntax in either the 'generic' sequential mode or the inherently multitasking FSM mode"

Pete is a programmer who specializes in embedded systems development. He can be contacted at petegray@ieee.org.

(continued from page 58)
different target. The default compiler mode is sequential. In addition, if the compiler is invoked without specifying a source, it begins an interactive session.

FSM-Specific Language Topics

FSM mode allows the use of Finite State Machine constructs, which are inherently multitasking. An application typically consists of multiple machines that — at any point in time — exist in a particular state. A good analogy would be that a machine is like a thread running in a process, or a machine is like a program running in a multitasking operating system.

The implementation of the FSM methodology requires that you list the allowable states of the machines in an application, defines the conditions whereby a machine state transition occurs, and declares the name and initial state of each machine. Each state and each machine has a unique name (they are, after all, identifiers).

State transitions are used to determine and execute a machine transition from one state to another, and achieve this through the assignment of the reserved word *nextstate*, optionally executing additional code during the transition.

Due to the nature of state machines, a transition may not include loops or calls (that is, anything that may cause a transition to "hang"). This apparent limitation really isn't limiting, rather, it proactively encourages you to produce code that is more appropriate to the state machine programming paradigm. The compiler automatically generates a high-speed, minimal overhead, context-switching mechanism based on an application's machine chain. This context switch examines the state transition conditions of each machine in a round-robin fashion, performing a machine state transition only when the transition conditions have been satisfied.

The demo version of the StatiC compiler has limited FSM capabilities — two machines and 12 states — which is enough to compile the FSM mode example program.

Program Structure: Sequential Mode

The structure of a typical sequential program looks like this:

```
Comments
  Directives
   Global Variable and Constant Declarations
    Procedure Block(s)
 Program Block
```

All items, except the Program Block, are optional. Comments can appear anywhere. Most Directives can appear anywhere. Global Variables and Constants must appear prior to being referenced (that is, referenced from a Procedure or the Program). Procedure Blocks must appear before the Program Block.

The Procedure and Program Block Structures follow. Optional elements are shown between square braces ([and]). Procedure Block Structure, see Example 1(a), define the name, parameters, and code of a callable routine. The Program Block Structure defines the code of the main program; see Example 1(b).

Listing One, a complete sequential mode StatiC program, performs simple terminal I/O and lets users turn LEDs on/off. The target system is New Micros's PlugaPod (http://www.newmicros.com/), which is based on Motorola's DSP56F803 digital-signal processor. This program displays instructions, then turns the LEDs on/off, depending upon what users type at the keyboard.

From within the program block, you see the statement:

```
word ichar 1
```

which declares a one-word variable, *ichar*, which is local to the program block. Next, the statement

```
^SCI0BR = 260    // baud 9600
```

loads the Serial Communications Interface (SCI) baud rate register with the value *260*, which sets the baud rate of the chip's SCI module to 9600. The statement works this way because I defined *SCI0BR*, near the top of the program, to be *$0F00*, the address (in Hex) of the baud-rate register for the PlugaPod, and I use the "^" operator. This could be thought of as meaning "load the contents of *$0F00* with 260." In StatiC, the same statement could have been written like this:

```
^$0F00 = 260
```

which would have achieved the same thing. However, it's good practice to substitute definitions for register addresses because the registers do not always have the same address within the same family of chips. Using definitions means that if you port your code to another chip, which doesn't have the same register address as the original, you'll only need to change the program in one place — in the *#define* directive. Besides, *SCI0BR* is a little more meaningful than *$0F00* to someone reading or maintaining the code.

The program then sets the various general-purpose input/output (GPIO) line-control registers, which are tied to the LEDs on the PlugaPod. Next, the statement:

```
call sci0output (@msg1) // display message
```

calls the SCI output routine, passing the address of *msg1* as the parameter. The constant *msg1* is a null-terminated string, and *sci0output* is coded to process the string passed to it, displaying the characters (via the SCI) to the terminal.

The program then enters a never-ending loop, reading the keystrokes and adjusting the LEDs accordingly. The statement:

```
call sci0input (@ichar)
```

calls the SCI input routine, passing the address of the local variable *ichar* as the parameter. The *sci0input* routine is coded to wait for keyboard input and return what was typed in the parameter passed to it.

Next, the character returned from the input routine is tested, and the LEDs are adjusted. The statement:

	Switch	Description
(a)		
	-a568	Produces assembler output for the a568 assembler.
	-a568cw	Produces assembler output suitable for use in CodeWarrior.
	-s$xxxx	Specifies the initial SP value, in Hex (defaults if unspecified).
	-f	Invokes the compiler in FSM mode.
	-b	Turns the bell on (for errors and/or warnings).
	-w	Displays warning messages.
	-m$xxxx	Specifies the initial SP value, in Hex, for machines.
(b)		
	-l	Lists the pseudocodes being generated, as comments, in the assembly output.
	-t	Produces a trace file of the compiler activity (ctrace.txt), including internal routine calls, parsing, and optimization details.

Table 1: *StatiC compiler switches.*

```
if ( ichar = '1' ) ^PADR = ^PADR | $0004 endif
```

performs a logical OR operation on the contents of the GPIO data register (*PADR* = Port A Data Register), if users type a "1" at the keyboard. ORing the data register with *$0004* sets bit 2 high, which turns the green LED on.

Finally, *sci0input* waits until the SCI status register (*SCI0SR*) indicates that a character has been entered, then puts the character into wherever *rchar* is pointing at:

```
ostat = ^SCI0SR
while ( ostat & $3000 ) <> $3000
   ostat = ^SCI0SR
endwhile
^rchar = ^SCI0DR
```

Recall that I passed *@ichar* to the routine, and the formal declaration of the routine was:

```
procedure sci0input (rchar)
```

so the statement:

```
^rchar = ^SCI0DR
```

actually stores the contents of the SCI data register (*SCI0DR*) into *ichar*.

Program Structure: FSM Mode

The structure of a typical FSM program looks like this:

```
Comments
  Directives
    Global Variable and Constant Declarations
 Procedure Block(s)
   State List
Transition Blocks(s)
  Machine Definitions
    Program Block
```

Many of the components of an FSM program structure are present in the sequential program structure. The extensions required for FSM mode are the State List, Transitions, and Machine Definitions, which must appear in order.

The State List simply lists the allowable application machine states:

```
statelist statename1 statename2 …
```

The Transition Block Structure defines the conditions required for a state change and the actions to perform when those conditions are met.

```
transition statename
begin
  condition expression
  causes
    statements
  endcondition
end
```

Finally, Machine Definitions lists the machines in the application, and defines their stack space and initial state. Each machine has its own stack space, and the compiler automatically initializes — and keeps track of — the stack pointer for each machine.

```
machine  machinename  stacksize  initial state
```

Listing Two, a complete FSM mode StatiC program, performs simple terminal I/O. Characters entered on the keyboard are received by the microcontroller and echoed on a PC running a terminal emulator. Again, the target system is NewMicros's PlugaPod, although this example also runs on the 805 chip (NewMicro's IsoPod), and — with modification to the SCI register addresses — the 807 (ServoPod).

First, notice that this application consists of two machines — *inputmachine* and *outputmachine*. The *main* part of the program, the Program Block:

```
^SCI0BR = 260 // baud rate 9600
^SCI0CR = 12  // 8N1
call sci0output (@msg1) // display
                        // welcome message
appstate = APPSTATEINPUT// the initial
                        //app state
```

sets up the Serial Communications Interface (SCI), displays a message, and sets the global variable *appstate* to be *APPSTATEINPUT*. This application is designed in such a way that the two machines are cooperative, and the setting of *appstate* determines their transition to/from one

(a)

```
procedure procedurename ( [parameter_list] )
begin
  [local variable declarations]
  statements
end
```

(b)

```
program
begin
  [local variable declarations]
  statements
end
```

Example 1: *(a) Procedure Block Structure; (b) Program Block Structure.*

state to another. Machines don't have to behave this way, but it's useful, for demonstration purposes.

Once the program block has been executed, all machines are activated. That is to say, they're put into their "initial state" as determined by the machine definition statements:

```
machine inputmachine 10 waitappinput
machine outputmachine 10 waitappoutput
```

inputmachine is put into *waitappinput* state, and *outputmachine* is put into *waitappoutput* state. Once in these states, they remain in these states until the state transition conditions have been satisfied. So, *inputmachine* is initially in *waitappinput* state, which is described in the transition block, thus:

```
transition waitappinput
begin
  condition appstate = APPSTATEINPUT
  causes
    nextstate = waitinput
  endcondition
end
```

However, *appstate* was defined as *APPSTATEINPUT* in the main program block, so the *inputmachine*'s transition condition is satisfied. This causes *inputmachine* to change states to *waitinput*.

Also, you'll notice that *outputmachine*'s initial state is *waitappoutput*, which is described in the transition block, thus:

```
transition waitappoutput
begin
  condition appstate = APPSTATEOUTPUT
  causes
    nextstate = doappoutput
  endcondition
end
```

Unlike *inputmachine*, *outputmachine*'s transition condition has not been satisfied, so no state change takes place, and *outputmachine* remains in the *waitappoutput* state.

At this point in time, *outputmachine* is waiting for its transition condition to be satisfied, and *inputmachine* has changed state to *waitinput*. So, looking at the *waitinput* transition block:

```
transition waitinput
begin
  condition ( ^SCI0SR & $3000 ) = $3000
  causes
    appchar = ^SCI0DR
    appstate = APPSTATEOUTPUT
    nextstate = waitappinput
  endcondition
end
```

inputmachine remains in this *waitinput* state until a keyboard key is pressed at the keyboard. The *outputmachine* is still waiting for its transition conditions to be satisfied.

When a key is pressed, *inputmachine*'s transition conditions are satisfied, a character is read from the SCI data buffer into the global variable *appchar*, the *appstate* is set to *APPSTATEOUTPUT*, and *inputmachine* performs a state change back to *waitappinput*.

"You simply create machines, as required, to perform the tasks you desire"

At this point, *outputmachine*'s state transition conditions have been satisfied (because appstate was set to *APPSTATEOUTPUT* by *inputmachine*), so *outputmachine* experiences a state change from *waitappoutput* to *doappoutput*. Looking at the *doappoutput* transition block:

```
transition doappoutput
begin
  condition ( ^SCI0SR & $C000 ) = $C000
  causes
    ^SCI0DR = appchar
    appstate = APPSTATEINPUT
    nextstate = waitappoutput
  endcondition
end
```

The *outputmachine* waits until the SCI is ready to send, then it loads the SCI data register with the global variables; *appchar*, sets the *appstate* to *APPSTATEINPUT*, and performs a state change back to *waitappoutput*. While this is all happening, *inputmachine* does nothing, because its state transition conditions have not been satisfied.

At this point in time, both machines are back in their initial states, and the whole cycle starts again.

Why FSM?

At this point, you may be wondering why anyone would want to code like this. The answer is because it's inherently multitasking. For example, say that you've coded the previous example and want to have the application monitor the PH level of the water in a fishtank (via the ADC), then set a GPIO line high (triggering an alarm) if the reading goes above a certain point. All you have to do is add another machine. Want to send PWM signals to activate a servo that opens a feeding tray? Add another machine.

There's no difficult "where do I put this new code so that the existing code still works?"—the machines run independently from each other (unless, of course, you deliberately design them to be cooperative). You could even run multiple machines on the same chip, which perform functions for more than one application; for instance, monitor a fishtank *and* monitor a home-security system.

You simply create machines, as required, to perform the tasks you desire. Each machine runs and changes state when its transition conditions are satisfied. All of the machines you define are running at the same time—the same as a multitasking operating system—and performing whatever function you've designed them to do. This is the true power of FSM programming.

Conclusion

My goal with StatiC is to create a dual-methodology language, which is easy to learn and use, yet advanced enough to perform multitasking in embedded environments. It had to be something that made rapid application development a reality, and not just an overused marketing phrase. But most of all, it had to be a language that I—as an experienced software developer—would want to use, as a matter of preference, over any other languages available in the domain. The StatiC language and compiler meet, and in some ways exceed, that goal. I'm surprised with what can be achieved using a relatively simple language—which just goes to show that sometimes the best solution to complex problems is a simple solution.

Acknowledgments

New Micros (http://www.newmicros.com/) produces inexpensive DSP56F80x microcontroller boards (IsoPod, ServoPod, MiniPod, and PlugaPod), as well as the JTAG cables. I'd like to thank Randy M. Dumse and Jack Crenshaw for their support and guidance. All compiler development was performed on a homemade P4 WXP box and an IBM 300PL running Linux RH9. I'm currently developing support for the Atmel AVR series of microcontrollers, as well as additional language features.

DDJ

Listing One

```
// port A definitions for GPIO (LEDs)
#define PAPUR    $0FB0
#define PADR     $0FB1
#define PADDR    $0FB2
#define PAPER    $0FB3
#define PAIAR    $0FB4
#define PAIENR   $0FB5
#define PAIPOLR $0FB6
#define PAIPR    $0FB7
#define PAIESR   $0FB8

// SCI0 definitions for terminal (RS232) interface
#define SCI0BR   $0F00
#define SCI0CR   $0F01
#define SCI0SR   $0F02
#define SCI0DR   $0F03

// constants - the welcome message
const msg1 "LEDs on/off 1/2=Green 3/4=Yellow 5/6=Red."
const msge 13,10,0

// output a null-terminated string to SCI0
procedure sci0output (optr)
begin
  word ostat 1

  while ^optr
    ostat = ^SCI0SR
    while ( ostat & $C000 ) <> $C000
      ostat = ^SCI0SR
    endwhile
    ^SCI0DR = ^optr
    optr = optr + 1
  endwhile
end

// read a character from SCI0
procedure sci0input (rchar)
begin
  word ostat 1

  ostat = ^SCI0SR
  while ( ostat & $3000 ) <> $3000
    ostat = ^SCI0SR
  endwhile
  ^rchar = ^SCI0DR
end

// the main program
program
begin
  word ichar 1

  ^SCI0BR = 260               // baud 9600
  ^SCI0CR = 12                // 8N1
  ^PAIAR = 0                  // enable LEDs
  ^PAIENR = 0
  ^PAIPOLR = 0
  ^PAIESR = 0
  ^PAPER = $00F8
  ^PADDR = $0007
  ^PAPUR = $00FF
  call sci0output (@msg1)    // display message
  ^PADR = 0                   // LEDs off
  while 1                     // loop forever
    call sci0input (@ichar)
    if ( ichar = '1' ) ^PADR = ^PADR ¦ $0004 endif
    if ( ichar = '2' ) ^PADR = ^PADR & $00FB endif
    if ( ichar = '3' ) ^PADR = ^PADR ¦ $0002 endif
    if ( ichar = '4' ) ^PADR = ^PADR & $00FD endif
    if ( ichar = '5' ) ^PADR = ^PADR ¦ $0001 endif
    if ( ichar = '6' ) ^PADR = ^PADR & $00FE endif
  endwhile

end
```

Listing Two

```
// definitions for SCI (RS232)
#define SCI0BR   $0F00
#define SCI0CR   $0F01
#define SCI0SR   $0F02
#define SCI0DR   $0F03

// global variables
word appstate 1
word appchar 1

// application control definitions
#define APPSTATEINPUT 1
#define APPSTATEOUTPUT 2

// constants
const msg1 "StatiC FSM SCI Demo Ready."
const msg2 13,10,0

// the application states
statelist waitappinput waitinput waitappoutput doappoutput

// the transitions
transition waitappinput
begin
  condition appstate = APPSTATEINPUT
  causes
    nextstate = waitinput
  endcondition
end

transition waitinput
begin
  condition ( ^SCI0SR & $3000 ) = $3000
  causes
    appchar = ^SCI0DR
    appstate = APPSTATEOUTPUT
    nextstate = waitappinput
  endcondition
end

transition waitappoutput
begin
  condition appstate = APPSTATEOUTPUT
  causes
    nextstate = doappoutput
  endcondition
end

transition doappoutput
begin
  condition ( ^SCI0SR & $C000 ) = $C000
  causes
    ^SCI0DR = appchar
    appstate = APPSTATEINPUT
    nextstate = waitappoutput
  endcondition
end

// define the machines
machine inputmachine 10 waitappinput
machine outputmachine 10 waitappoutput

// a procedure used at start-up, to display welcome message
procedure sci0output (optr)
begin
  word ostat 1

  while ^optr
    ostat = ^SCI0SR
    while ( ostat & $C000 ) <> $C000
      ostat = ^SCI0SR
    endwhile
    ^SCI0DR = ^optr
    optr = optr + 1
  endwhile
end

// the main program
program
begin
  ^SCI0BR = 260                 // baud rate 9600
  ^SCI0CR = 12                  // 8N1
  call sci0output (@msg1)       // display welcome message
  appstate = APPSTATEINPUT      // the initial app state
end

// at this point, all of the defined machines are 'running'
```

DDJ

Building on TiVo

Extending the home media platform—legally!

ARTHUR VAN HOFF
AND ADAM DOPPELT

TiVo is a digital video recording (DVR) device and service that, among other things, lets you store up to 140 hours of television programming, pause/rewind/fast-forward live TV shows, find and record shows with your favorite actor, and record one program while watching another. It's no surprise that TiVo's success has made it an attractive target for hobbyists and hackers, as witnessed by the Internet sites, books, and hardware extensions available to extend and improve the user experience. Though customizing your digital video recorder is not encouraged or endorsed by TiVo, we nonetheless recognize there is a desire to extend the experience.

In this article, we introduce a new technology for doing just that. Currently available as a developer preview on standalone TiVo devices, the Home Media Engine (HME) lets you build applications that integrate seamlessly with the TiVo user experience. The HME technology originated at Strangeberry, a Silicon Valley startup we founded, which was acquired by TiVo in early 2004.

HME enables a wide variety of new application types and lays a solid foundation for integrating third-party functionality into the TiVo user interface. Dust off your compilers, because with HME, you can finally legally hack your TiVo DVR!

Arthur is a principle engineer at TiVo and Adam is one of the architects for the HME project. They can be contacted at avh@tivo.com and amd@tivo.com, respectively.

Rewind

When we started Strangeberry in 2002, we had just completed lengthy cubicle stints writing enterprise software and were eager to turn our attention to something near and dear to our hearts— consumer electronics. Since the Strangeberry founders were TiVo enthusiasts, we investigated developing the first third-party aftermarket application for TiVo DVRs. We built a concept application in PowerPoint and pitched it to TiVo representatives. They loved the idea, but it quickly became clear that it would never see the light of day. Back then, integrating with TiVo DVRs from the outside turned out to be technically challenging with limited business opportunities.

Frustrated, we decided that the market needed an open DVR and set out to build one from scratch. Within a couple of months, we created a reasonably functional system based entirely on open-source software and featuring powerful user interface (UI) primitives.

Our experiments generated clouds of speculative smoke, but we couldn't quite find the fire. We tried to imagine how future components would be connected together. Today's devices lack connective intelligence and require too many remote controls. What kind of connector will you find on the back of your television in 10 years? NTSC? Coax? Ethernet? Wireless?

> "HME is currently available to developers for building applications for the TiVo platform"

We investigated emerging Standards such as Universal Plug-and-Play (UPnP), which allows access to network-based media resources. However, it is inflexible and hard to extend in meaningful ways. Besides, we don't want our media to show

up in someone else's user interface. Instead, we want our media in our user interface on someone else's device! (We want iTunes, not Windows Explorer.)

It quickly became clear that the architecture we created solved many of these problems. We decided to focus on building a Digital Media Receiver based on this technology, which we dubbed the "Home Media Engine" (HME).

The Strangeberry Digital Media Receiver was able to display rich multimedia UIs for network-based or embedded applications on television screens. Developers could create attractive applications, with alpha-blended graphics and smooth animations. We had built a truly open technology that let developers control the user experience and integrate diverse applications on a single television screen.

Now Playing

HME is currently available to developers for building applications for the TiVo platform. Figure 1 describes the TiVo platform. The initial release of HME is a developer preview that makes it possible to build PC-based applications that add interesting new functionality to the TiVo box. HME developers have already built personal media applications for music and photos, RSS readers, games, weather viewers, news tickers, search engines, and much more.

For techies like us, it is easiest to think of HME as X/11 for television. Instead of the tired old X primitives, HME supports image compositing, transparency, animation, antialiased text, streaming media, and remote-control input.

HME applications advertise themselves using the Rendezvous service discovery protocol. TiVo devices on the local network will discover these applications and add them to the Music & Photos screen in the TiVo UI (see Figure 2). When users select an HME application, the TiVo device connects to it and the application can display its UI on the television screen. Remote-control input is routed back to the application, giving it full control over the user experience.

The display protocol is simple but effective. The application builds a hierarchical tree of views, each of which can contain a resource, such as a color, image, or text string. The device renders the tree and combines text, graphics, and video into a UI. When users press a key on the remote, an event is delivered to the application, which can then adjust the UI accordingly.

HME has features that let applications create a rich user experience. It contains animation primitives that allow for smooth transitions and fades. Assets can be loaded directly from the application or from an external location specified by a URL.

Pause

The hardware inside the TiVo DVR superficially resembles a modern PC, but with reduced cost comes reduced horsepower. Most TiVo DVRs are built around a 180MHz MIPS CPU with 32MB of memory and limited memory bandwidth. The system is continuously recording and playing back video, which consumes a sizable chunk of the system's limited resources.

Luckily, each TiVo device contains a graphics chip that provides a reasonably powerful 2D compositing engine. The chip is a fascinating component with interesting potential and equally interesting restrictions. To work around these restrictions, the HME team built a powerful tiling engine that breaks the HME view hierarchy into a list of regions that can be rendered using the graphics hardware.

The tiling engine, informally known as the "supercollider," detects situations that would violate the constraints imposed by the graphics chip. When the graphics chip becomes overloaded, portions of the user interface are rendered by the main CPU. As the load increases, the rendering engine continues to paint the screen correctly, but performance gradually degrades, similar to a modern rendering pipeline. The result is a smooth and interactive user experience with an intuitive API for developers.

Thumbs Up

Because the HME is currently only available as a developer preview release, users need to enter a secret backdoor key sequence to turn on support for the HME rendering engine. In future releases, the backdoor will be removed and HME turned on by default.

The HME developer release is freely available and includes a Java SDK, HME simulator, sample applications, documentation, and lots of source code (http://www.tivo.com/developer/). The SDK is available under the CPL license (similar to LGPL) and can be included in commercial products without requiring a separate license.

HME is part of the 7.1 release of the TiVo System software and available on all TiVo Series2 standalone units. Standalone units require a USB Ethernet or wireless network adapter to connect to the local network.

What is the secret backdoor key sequence? Go to the System Information screen and enter clear-clear-0-0 using the remote. You will notice that the Music & Photos screen is renamed to Music,

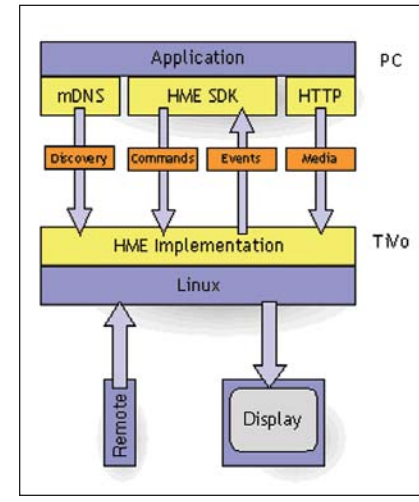


Figure 1: *TiVo architecture.*

Figure 2: *TiVo Music screen.*

Photos & More. HME is now enabled on the TiVo box.

Enter

To get started with HME, we present in Listing One TiVo's version of the ubiquitous HelloWorld application. HelloWorld creates a text resource by specifying the font, color, and a text string. The text resource is placed into the application's root view and appears centered on the television screen.

Since HelloWorld is easy to understand but a bit dull, we'll explore another sample application that is part of the SDK—TicTacToe. The TicTacToe (see Figure 3) example covers simple screen layout, animation, remote control events, and sound effects.

Screen Layout

The *init* method is the main entry point for each application, and one of its primary tasks is to layout the screen. Listing Two is TicTacToe's *init* method.

"Views" are the basic building blocks of HME applications and provide clipping, a coordinate system, and transparency, but are invisible unless they contain resources. "Resources" are concrete elements that are drawn inside views. The layout of the screen is controlled using the view hierarchy. Each view can contain many child views but it can contain only a single resource.

Each application has a root view. The root view can be thought of as the television's "desktop." The size of the root view is 640×480 pixels. However, application developers should be careful to use only the "title-safe" area of the screen. Older televisions have a black border of paint around the edge that can obscure almost 32 pixels on each side!

TicTacToe's *init* method creates two views as children of the root view. The *piecesView* is initially empty and will eventually contain the X and O pieces used throughout the game. Later, we will add children to *piecesView,* one for each move made in the game. The grid view paints the TicTacToe grid that serves as the playing field. It is placed in a specific location on screen and populated

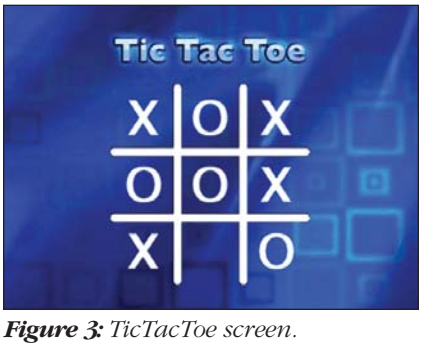


Figure 3: *TicTacToe screen.*

with the grid.png image contained in the application.

Layering

Because *piecesView* is created before grid, *piecesView* and all of its children will paint before grid. On screen, this has the effect of making the pieces appear "underneath" the grid. This is important because when the game ends, we want to highlight and animate the pieces in various ways and the effects are improved if the pieces are underneath the grid. In fact, this is the only reason that *piecesView* exists at all—to provide layering in a specific way that makes the special effects look more dramatic.

The images array contains the X and O text resources that are used to paint the pieces. A single resource can be used in many different views, and we only need a single X and O for the entire lifetime of the application.

Keys

Now that we've put a grid on screen, we need to respond to remote-control events to actually play the game when the user presses buttons. Listing Three is TicTacToe's *handleKeyPress* method. TicTacToe uses the numeric pad on the TiVo remote control. The one button (KEY_NUM1) represents the upper-left corner of the grid, and the nine button (KEY_NUM9) is the lower right. If users hit a numeric key on the remote, we convert it to x-/y-coordinates and make a move. If a user hits any other key, we play the bonk.snd error sound. Nearly all of our standard set of TiVo keys and sound effects are available for use by HME applications.

Animation

Finally, TicTacToe takes advantage of HME's powerful animation primitives to provide feedback when the game is over. By default, changes made to HME views using accessor methods, such as *setBounds*, *setTransparency*, and *setScale,* occur instantly. But these same view methods accept an optional animation argument. By specifying an animation, the change can be performed incrementally over some period of time. Listing Four is a snippet from the game-over code.

When a player wins, the X and O pieces appear to explode away from the board over a period of one second. The pieces gradually fade away at the same time. If the game is a draw, the pieces fade without the satisfying explosion, leaving deadlocked players to stare at the stark lines of the game grid.

HME was designed from the ground up to produce an NFL-quality experience, not an HTML experience. Even a simple game like TicTacToe benefits greatly from the pizzazz offered by HME.

> "By open sourcing the SDK, we hope to tap into a vibrant community and help developers build on our position"

Fast Forward

TiVo is excited about opening up the platform to developers. We can't wait to see what applications you'll write. By open sourcing the SDK, we hope to tap into a vibrant community and help developers build on our position at the center of the modern living room. This is your official invitation to join the party!

To encourage your creativity, we're hosting the TiVo Developer Challenge, where you can compete for fame and fortune by writing the most creative and interesting HME applications. The contest has great prizes and a jury consisting of well-known industry leaders. We hope it will inspire many of you to develop the next killer app for the TiVo DVR. For competition details, see http://www.tivo.com/developer/.

This initial release is just the tip of the iceberg for the TiVo platform. In upcoming releases, HME will be enabled by default so that all TiVo service users can enjoy the benefits of an open platform. We plan to introduce additional deployment models that don't require a PC, and in some cases, don't even require a home network. The HME API will provide access to more DVR functionality, including scheduling, recording, and playback. Stay tuned—you don't want to miss the show.

DDJ

Listing One

```
import com.tivo.hme.sdk.*;
public class HelloWorld extends Application {
    protected void init(Context context) {
        root.setResource(createText("default-36-bold.font",
                                    "0xffffff", "Hello, world!"));
    }
}
```

Listing Two

```
public class TicTacToe extends Application
{
    View piecesView;
    Resource images[] = new Resource[2];
    int gridX, gridY;
    ...
    protected void init(Context context)
    {
        piecesView = new View(root, 0, 0, width, height);
        gridX = (width  - 300) / 2;
        gridY = (height - 300) / 2;
        View grid = new View(root, gridX, gridY, 300, 300);
        grid.setResource("grid.png");
        images[0] = createText("default-72-bold.font", "0xffffff", "X");
        images[1] = createText("default-72-bold.font", "0xffffff", "O");
    }
...
```

Listing Three

```
public boolean handleKeyPress(int code, long rawcode)
{
    if (code >= KEY_NUM1 && code <= KEY_NUM9) {
        int pos = code - KEY_NUM1;
        makeMove(pos % 3, pos / 3);
        return true;
    }
    play("bonk.snd");
    return false;
}
```

Listing Four

```
Resource animation = getResource("*1000");
for (int x = 0; x < 3; ++x) {
  for (int y = 0; y < 3; ++y) {
    View v = pieces[x][y];
    if (v != null) {
      if (victory) {
        v.setLocation(v.x + (x - 1) * 400, v.y + (y - 1) * 300, animation);
      }
      v.setTransparency(1, animation);
        ...
    }
  }
}
```

DDJ

Adding Diagnostics to .NET Code

How you can use Framework classes to add diagnostic messages to your code

RICHARD GRIMES

The .NET Framework designers recognized that you'd want to add trace messages liberally in your code. However, since trace messages are only useful for debug builds, the designers had to provide a mechanism so that you would not have to edit your code to remove trace messages from release builds. C# has conditional compilation similar to native C++, where the compiler only compiles a section of code depending on whether a specific symbol is defined (see Listing One). However, this usually results in code that is unreadable. Instead, C# (and VB.NET, but not Managed C++) has support for the *[Conditional]* attribute, which is applied to methods in libraries and specifies a symbol that must be defined to use the method. Another assembly can have code that calls this method, but the compiler only compiles this call into the final assembly if the symbol is defined.

Contrast Listing Two with Listing One. Listing One uses conditional compilation so that the code only appears in the assembly (app.exe) if the symbol DEBUG has been defined. Listing Two shows fragments of code from two assemblies; all of the code in the first half of the listing is compiled into the first assembly (lib.dll) regardless of whether the DEBUG symbol is defined. The method *PrintDebug* is added to the assembly and is marked with the *[Conditional]* attribute. The second fragment of code has a call to this method, but the compiler only adds this to the assembly if the symbol DEBUG is defined. Listing Two is definitely preferable to the code in Listing One; however, note the subtle difference between the two when DEBUG is not defined. In Listing One, the conditional code is not compiled at all; in Listing Two, the conditional code is compiled, but it is not called. The *[Conditional]* attribute is acted upon by compilers, but the Managed C++ compiler does not recognize this attribute, so you have to use conditional compilation. Listing Three shows how to use macros to access a conditional method.

Richard is the author of Programming with Managed Extensions for Microsoft Visual C++ .NET 2003 *(Microsoft Press, 2003). He can be contacted at richard@richardgrimes.com.*

Trace Messages

Why am I saying this? Well, the Framework library provides two identical classes—*Debug* and *Trace*—which are used to provide traces and asserts. These classes only differ in that the methods of *Debug* are marked with *[Conditional("DEBUG")]*, whereas the methods of *Trace* are marked with *[Conditional("TRACE")]*. Visual Studio.NET 2003 C# and VB.NET projects define both *DEBUG* and *TRACE* for Debug builds and *TRACE* for Release builds. Review what I have just said: By default, the methods of *Trace* can be called in Release builds.

By themselves, the methods of the *Trace* and *Debug* classes do very little—they only accept some diagnostic message from your code, and do a little formatting on them. Reporting that message is the work of a trace listener, a class that derives from the abstract class *TraceListener*. By default, every application domain

"The Framework library provides two identical classes—*Debug* and *Trace*—which are used to provide traces and asserts"

in a process is initialized with a single trace listener of type *DefaultTraceListener*. Each application domain will have a collection of trace listeners that you can access through the static *Listeners* property on either the *Trace* or *Debug* class. You can use this collection to add or remove listeners in your code. When an application domain is created, the runtime will read the configuration file for the application and add or remove the listeners mentioned there. Listing Four shows a sample configuration file, which indicates that when an application domain is created, it should not have the default trace listener. Instead, however, it should have an instance of *TextWriterTraceListener* to log the trace messages to the file mentioned in the *initializeData* attribute.

What should you trace? There are two main types of data: First, you may want to test code coverage, so it is useful to record each method called and to record

(continued from page 68)
the branch of a test statement. The second type of data are metrics of how well your algorithms are working, so you may decide to trace input parameters, results, and selective intermediate values. The problem, of course, is that this represents a large amount of data and the Framework classes provide few tools to help you. To trace a message, you call either the *WriteLine* or *Write* method. These methods are overloaded, but in effect, you can trace just a message string, or a message string and a category string. The category string is just a description — it has no effect on how the trace listener handles the trace message.

There are versions of *Write* and *WriteLine* that take a Boolean parameter; these are called *WriteIf* and *WriteLineIf*. Only if this parameter is True is the method called. Conceivably, you could create a global collection of switches and use this to allow only messages of a specific category to be traced. One suggestion about how to do this is shown in Listing Five — the *MySwitches* class has public static members so that they can be accessed throughout the assembly. These members indicate if messages of a particular category (in this case, for code that obtains, analyzes, and presents data) should be traced.

The diagnostics namespace contains an abstract class called *Switch* that can be used to obtain a value from the <*switches*> element in the application configuration file. There are two classes in the Framework derived from *Switch*: *BooleanSwitch* is passed the name of a switch in its constructor, and if the switch value is nonzero, it sets the *Enabled* property to True; *TraceSwitch* goes one step further — it returns the value of the switch in the *Level Property* as one of the values in the *TraceLevel* enumeration. Listing Six shows how to use the *BooleanSwitch*. There are two parameters to the *BooleanSwitch* constructor; the second parameter is not used, it is left over from the first beta of .NET.

Trace Listeners

When you generate a trace message, either through *Trace* or *Debug*, the *TraceInternal* class is called. *TraceInternal* will access the application domain's *Listeners* collection and iterate calling an appropriate method on each trace listener. If you call *Debug.Write(String)*, then *TraceInternal* will call *TraceListener.Write(String)* on each *TraceListener* in the collection. If you call the conditional methods (*WriteIf* or *WriteLineIf*), then it is the *TraceInternal* equivalent of these methods that performs the test on the Boolean parameter.

If you call one of the overloads that take a category string, then the trace listener decides what the category parameter means and how to react to it. The implementation of these methods in the base class, *TraceListener*, merely concatenates the two strings separated by a colon. These methods are virtual, so it is possible that a trace listener could behave differently for each category (for example, it could log only specific categories); however, all of the Framework trace listener classes use the base class implementation.

Asserts

As the name suggests, an assert tests that some condition is True, and if the test fails, the user is informed. Asserts should be used on important conditions where a failure will mean that the code will fault. The default trace listener class recognizes this and indicates that an assert has failed with a modal dialog that you cannot ignore. It is important that you do not let asserts be active in release builds. An assert tells your users that not only is there a bug in your code, but that you thought that there was one and you have released the code without fixing it! Quite rightly, the *Debug* class has an *Assert* method, because the only situation when you should use an assert is in a debug build. Again, however, *Trace* has the same methods as *Debug*, but you must not be tempted to call *Assert* on this class. The reason is that by default, C# and VB.NET projects in Visual Studio .NET define the TRACE symbol in release builds, which enables you to call *Assert* in release builds.

There are three overloads of the *Assert* method: The most flexible has the

condition to check a short message and a long message; the other two have just the short message or no message at all. If you don't provide a message when you receive a failed assert at runtime, you will have no idea about the failure condition. The *Fail* method does not take a conditional parameter, but can take either a short message or a short and a long message. Clearly you have to perform some logic to determine whether *Fail* should be called. However, it is useful during development to put a call to *Fail* in code that has not been completely implemented; for example, when you add an interface to a class, it is good practice to put a call to *Fail* in each method body. *Fail* is marked with *[Conditional]*, and like *Assert*, you should only call it through the *Debug* class.

All the overloads of *Assert* and *Fail* call an overload of *TraceInternals.Fail* that takes two string parameters; this method iterates through the trace listeners collection calling the *Fail(string, string)* method on each one. The base class implementation of this method merely appends the long message and the short message to a failure message, and passes this to the trace listener's *WriteLine* method. This is used by all of the trace listeners except the *DefaultTraceListener* class. The *DefaultTraceListener* class displays the assert using a modal dialog, so it is important that you do not let this be shown in code that runs in a service or on a remote server machine. A modal dialog blocks the current thread until the dialog is dismissed and, thus, it must only be used in a situation when there is a human who can dismiss the dialog. Again, the best guard against this is to only call *Assert* (or *Fail*) on *Debug* so that it can only be called in debug builds when, presumably, you have also marked the service to interact with the desktop.

The code handling *Assert* (and *Fail*) has more problems. In Figure 1, an assert dialog, the short message, and the long message are displayed on separate lines, which are followed by a stack trace. If the project was compiled with debug symbols (which will be the case for a debug build) and the symbols are available, then the stack trace gives the name of the file and the line number where the method is defined. If the symbols are not available, then the stack trace will not have filename and line number information.

There are some more points to be made about the dialog shown in Figure 1. The names of the buttons are Abort, Retry, and Ignore. The dialog caption gives a helpful hint that Abort means you should quit the application, Retry means that the debugger should attach to the process, and Ignore means the application should continue executing. Why didn't the .NET team use the names Quit, Debug, and Continue on the buttons? The reason is that the dialog is a standard Windows dialog shown by the Win32 *MessageBox* function, and this function does not let you give custom names to the buttons. Another option would be to use Windows Forms to create a dialog; however, this would mean that every application that uses the diagnostic classes would have a dependency on the *system.windows.forms* assembly.

If you click Abort, then it just means that the application ends and the assert will not be reported to other trace listeners (which is what you need, because you will want to know exactly what has happened). The only way around this issue is to use either the configuration file, or to programmatically remove the *DefaultTraceListener* that the system has added to the *Listeners* collection (called *Default*) and add it to the end of the collection.

If you call *Debug.Assert* in Managed C++, you discover that no stack trace is shown. The reason is that the stack trace has the name of each method in the stack, and to do this, the *DefaultTraceListener* attempts to get the method name and the type of the class. However, in Managed C++, the entry point will be a global method. This means that the type will be null and the code does not check for this—it merely bails out giving an empty string for the entire stack trace. If you add asserts to a process assembly in Managed C++, it is prudent to include the method name and class in the long description. The alternative is to write your own trace listener that fixes this problem.

Default Trace Listener

The Framework library has three trace listeners; see Table 1. *DefaultTraceListener* sends trace messages to *OutputDebugString*, *Debugger.Log*, and a file. *OutputDebugString* works by throwing the system exception (SEH) 0x40010006, passing the length of the string and the string as exception parameters. Listing Seven shows this: The call to *OutputDebugString* and *RaiseException* do the same thing. Windows catches this exception and makes this data available through the shared memory-mapped file *DBWIN_BUFFER*. Because this buffer is shared, it must be protected from multithreaded access; in particular, only one thread must be able to write to the memory-mapped file at any one time. To do this, the system creates two events—you can find out how these are used by looking at the dbmon example in the Platform SDK. The problem is that *OutputDebugString* blocks until the process reading the memory-mapped file has set one of these events to indicate it has read the data. In other words, *OutputDebugString* couples your process to another process that will read the debug string. This is bad news, so you should not generate *OutputDebugString* messages in release code. Again, this means that you should only generate trace messages through the *Debug* class in debug builds

DefaultTraceListener will also log a trace message to the *Debugger.Log* class, which communicates with an attached debugger. If the Visual Studio .NET debugger is attached to a process that generates a trace message handled by *DefaultTraceListener*, you will see the message appear in the Output window. In addition, you can specify that the

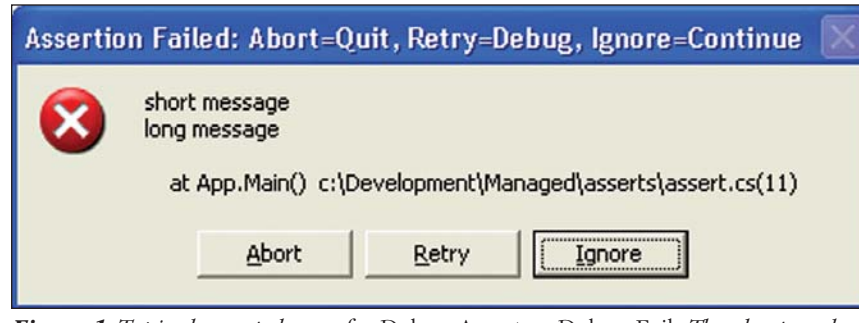


Figure 1: *Typical assert shown for* Debug.Assert *or* Debug.Fail. *The short and long descriptions are shown on separate lines followed by a stack trace.*

Class	Description
DefaultTraceListener	Sends the message to OutputDebugString, Debugger.Log, and to a log file for trace messages and Fail. In addition, a modal dialog is displayed for Fail.
EventLogTraceListener	Asserts and trace messages are written to an event log whose source is specified by the constructor.
TextWriterTraceListener	Writes asserts and trace messages to a TextWriter, Stream, or a file.

Table 1: *Trace Listeners in .NET Framework 1.0 and 1.1.*

message should be appended to a file. To do this, all you have to do is set the *LogFileName* property to the name of the file, which you can only do programmatically. When a trace message is generated, *DefaultTraceListener* opens your logging file, writes the message, and then closes the file; no synchronization is used. So if you have two threads generating trace messages, there is a small chance that one thread will have the file open while the other thread attempts to open the file; this will result in a file access fault and a .NET exception. This is tolerable during testing because you know what is causing the issue, but this should not happen in release mode.

Again, the *Fail* method, shows a modal dialog. In release builds, you should never show an assert dialog, so you should never let *Assert* or *Fail* be called in release builds. Even in debug builds you should not allow a modal dialog to be shown in a service. You can prevent this by setting the *AssertUiEnabled* property to False, which can be done either programmatically or through the configuration file.

Event Log Trace Listener

EventLogTraceListener derives from *TraceListener*, but overrides very little. There are three constructors, and the one that is called through the configuration file takes a single string, which is the name of the event log source. This means that all messages are sent to the Application log, with the potential detrimental effect of swamping the log with trivial messages so that you cannot see events generated by other processes. The other constructors let you programmatically specify the log and source.

Once the event log is open, it remains open until the trace listener is disposed, or when it is explicitly closed by calling *Close*. Both trace messages and asserts are handled by writing an event log entry. However, no attempt is made to specify the event log entry type (for example, an assert should be an *Error* type). The implementation of the *EventLog* class in Version 1.1 of the .NET Framework is broken because it does not let you log an event using an event log message file, which means that localization is the responsibility of the code generating the event rather than the code displaying the event (which will know what language you would prefer to use to read the messages). All other processes on Windows generate events that are localized by the event log viewer; since .NET deems that it should be different to all other processes, I regard it as a broken implementation. For all of these reasons, I advise you to not use *EventLogTraceListener* (or even the *EventLog* class) in release builds, and preferably not to use it at all.

TextWriter Trace Listener

The *TextWriterTraceListener* class lets you attach any stream to the trace listener. There are several overloaded constructors, and essentially, these let you create the trace listener from an open *TextWriter* object or an open stream, or you can pass the name of a file and the constructor attempts to open that file (this is the constructor that is called if you specify this trace listener in the configuration file). The trace listener's *TextWriter* will remain open until the trace listener is disposed or is closed explicitly by calling *Close*. However, this is a problem if you use the configuration file because every application domain attempts to create the trace listeners identified.

For example, Listing Eight shows the configuration file that specifies that the process should log to a file called "logFile.log." The first application domain to be created creates an instance of *TextWriterTraceListener*, opens this file for exclusive access, and keeps it open. The second application domain creates another instance of *TextWriterTraceListener* and attempts to open the same file — this call fails. Listing Nine shows a version of *TextWriterTraceListener* that is safe to use in processes that have multiple application domains and Listing Ten shows the appropriate configuration file.

If you use this trace listener, it is important that you are aware of *Flush*. Some streams are buffered and so the data is not written out until the *Flush* method is called. This method is implemented on *TraceListener* as well as *Debug* and *Trace*. If you call code that is likely to throw an exception, it is important to call *Flush* before you call this code, or at least call *Flush* in your exception handler. Asserts are a problem because a failed assert will not throw an exception. In this case, you should set the *AutoFlush* property of *Debug* to True or use the *autoflush* attribute of the <*trace*> element in the configuration file.

Wrap Up

Postmortem diagnostics are extremely important to determine why your code has failed, so traces and asserts are very important in debug builds. However, there are numerous issues with the Framework diagnostic classes, and for all of these reasons, you should not let diagnostic messages be generated in release builds. Never call *Assert* in release builds, avoid using *EventLogTraceListener* in release or debug builds (because it abuses the event log), and if you want to use *TextWriterTraceListener,* use the class in Listing Nine to prevent issues in processes that have more than one application domain.

DDJ

Listing One

```
   // Assembly app.exe
   int i = 42;
#if DEBUG
   Console.WriteLine(
      "initial value of i is " + i.ToString());
#endif // DEBUG
```

Listing Two

```
// Assembly lib.dll
class DbgCode
{
   [Conditional("DEBUG")]
   public static void PrintDebug(string str)
   {
      Console.WriteLine();
   }
}
// Assembly uselib.exe
int i = 42;
// Next line will compile only if this assembly is compiled with DEBUG defined
DbgCode.PrintDebug("initial value of i is " + i.ToString());
```

Listing Three

```
// Call PRINTDEBUG rather than PrintDebug
#ifdef DEBUG
   #define PRINTDEBUG DbgCode::PrintDebug
#else
   #define PRINTDEBUG __noop
#endif
```

Listing Four

```
<configuration>
 <system.diagnostics>
  <trace autoflush="true">
   <listeners>
    <remove name="Default"/>
    <add name="myListener"
     type="System.Diagnostics.TextWriterTraceListener"
     initializeData="MyListener.txt"/>
   </listeners>
  </trace>
 </system.diagnostics>
</configuration>
```

Listing Five

```
class MySwitches
{
   public static bool DataAcquisition {get;}
   public static bool DataAnalysis {get;}
   public static bool Presentation {get;}
   // Implementation omitted
}
```

```
// Use switches
WriteLineIf(MySwitches.DataAnalysis, "Searching for user...");
```

Listing Six

```
<configuration>
   <system.diagnostics>
      <switches>
         <add name="DataAnalysis" value="1"/>
      </switches>
   </system.diagnostics>
</configuration>
// MySwitches constructor
static MySwitches()
{
   BooleanSwitch data = new BooleanSwitch(
    "DataAnalysis", "this parameter is not used");
   dataAnalysis = data.Enabled;
   // other code
}
```

Listing Seven

```
// Unmanaged C++
LPCSTR str = "Test String\n";
OutputDebugString(str);
LPCSTR args[2] = {reinterpret_cast<LPCSTR>(strlen(str)), str};
RaiseException(0x40010006, 0, 2, reinterpret_cast<const DWORD*>(args));
```

Listing Eight

```
<configuration>
 <system.diagnostics>
  <trace autoflush="true">
   <listeners>
    <add name="myListener"
     type="System.Diagnostics.TextWriterTraceListener"
     initializeData="logFile.log"/>
   </listeners>
  </trace>
 </system.diagnostics>
</configuration>
```

Listing Nine

```
// Defined in assembly domainsafe.dll
class DomainSafeTextTrace : TextWriterTraceListener
{
   public DomainSafeTextTrace()
   {
      Initialize("Trace");
   }
   // The parameter is the base name for the log file.
   public DomainSafeTextTrace(string str)
   {
      Initialize(str);
   }
   // Create a file with a name that includes the name of application domain.
   //  This will be unique for each application domain.
   protected void Initialize(string str)
   {
      this.Name = "DomainSafeTextTrace";
      string strFile = str + AppDomain.CurrentDomain.FriendlyName + ".log";
      this.Writer = new StreamWriter(strFile, true);
   }
}
```

Listing Ten

```
<configuration>
 <system.diagnostics>
  <trace autoflush="true">
   <listeners>
    <add name="myListener"
      type="DomainSafeTextTrace, domainsafe"
      initializeData="domainSafeLog"/>
   </listeners>
  </trace>
 </system.diagnostics>
</configuration>
```

DDJ

Reducing the Size of .NET Applications

Smaller EXEs without native code

VASIAN CEPA

Executable files for the .NET Framework currently cannot be packed by binary file compressors such as UPX (http://upx.sourceforge.net/) because .NET uses customized sections in the Portable Executable (PE) file (which is used by all Windows executable files). The .NET Execution Engine expects Common Language Infrastructure (CLI) data to be in the proper sections of the PE file. However, CLI data is placed in the PE sections uncompressed by default.

In this article, I present a technique for reducing the size of .NET executables without using native code or otherwise modifying the PE format. Instead, I use reflection, which is supported by the .NET Framework, and pack the applications at a higher level.

Reducing the size of applications has several benefits:

- The disk space required is smaller. While disk space is usually not a problem in desktop computers, it can be in portable devices that run .NET Framework.
- Smaller executables load faster because of fewer disk accesses. Even if you uncompress the data in memory, RAM access is very fast and compressed executables still load faster than the original uncompressed ones.
- Using compression combined with, say, in-memory encryption/decryption makes it harder to disassemble .NET applications. This helps protect intellectual property.

Vasian is a Ph.D. candidate in the Darmstadt University of Technology's Software Technology Group. He can be contacted at cepa@informatik.tu-darmstadt.de.

The technique I present to compress .NET executables can be used with the main executable (EXE) file of .NET applications and with .NET DLL files that follow the .NET XCOPY paradigm. The technique will not work for DLL files placed in the Global Assembly Cache (GAC), which can be shared system wide or for DLL files shared by more than one application that is not aware of this technique.

The technique does not affect the usual development of .NET applications. The application EXE file and DLL files are built and compiled as usual. The technique can be applied as an additional step after you build the release version. Because of the generality of the solution, it is possible to generalize the technique to work with generic EXE and DLL files written in any .NET front-end language. I have created a tool called .NETZ, which is based on this technique (source code and binaries are available at http://www.st.informatik.tu-darmstadt.de/static/staff/Cepa/tools/netz/index.html and from *DDJ*; see "Resource Center," page 5). Here, I explain how .NETZ works, giving examples of the most interesting points. This makes it easier to apply customized versions of this technique in .NET applications.

Selecting a Compression Library

To compress the .NET executable data, you need a compression library. I use the open-source #ZipLib (http://www.icsharpcode.net/), which implements various compression algorithms. I use only the usual ZIP format (http://www.pkware.com/products/enterprise/white_papers/appnote.txt) from this library. To compress the data, any third-party ZIP tool will do. For example, *pkzip25— add app.zip app.exe* can be used in a batch file and pack app.exe as app.zip. The resulting app.zip is about 60 percent smaller on average than the original. .NETZ automates this step and does zipping programmatically using #ZipLib. To unzip the application at runtime, you need access to the unzip code. This means that you have to distribute the zip library with the compressed executable file. The size of #ZipLib is about 115KB. But given that it is open source, you can remove from it support for all compression formats other than ZIP. Moreover, you only need to leave the unzip code. If you do this, the size of the compiled zip.dll you need to distribute with the application becomes about 60KB. This is the only size overhead of this method. For applications larger than 200KB, however, you still gain size when compressing. You can do better by using compression libraries written especially for this technique.

> "Smaller executables load faster because of fewer disk accesses"

The Starter Application

The heart of the technique is a small starter application (stater.exe), which unpacks the data in memory and starts a packed application. (The source code file starter.cs is also available electronically.) Figure 1 illustrates how the starter application handles .NET EXE files. Keep in mind that the goal here is to create a packed application that, apart from size, is undistinguishable to users from the original application. For this reason, I pack the app.zip data as part of the starter application. The easiest way to do this in .NET is to pack the data as a resource. The resources of .NET applications are packed along with the application in the same physical executable file. Listing One produces a valid resource file. While any name will do, I've named the resource "app.exe" so I can access it later in the starter application. In the starter application at runtime, you first access the packed resource like this:

(continued from page 74)

```
ResourceManager rm =
            new ResourceManager ("app",
Assembly).GetExecutingAssembly()0
      (byte[])rm.GetObject("app.exe");
```

Then you unzip the data in memory:

```
string zipPath = "app.exe";
MemoryStream zipFile =
                  new MemoryStream(data);
ZipFile zf = new ZipFile(zipFile);
ZipEntry ze = zf.GetEntry(zipPath);
Stream zs = zf.GetInputStream(ze);
byte[] uzdata = new byte[ze.Size];
sz.Read(uzdata, 0, uzdata.Length);
```

This code is specific to #ZipLib, and the *zipPath* value is unique to the ZIP file format. The zip entry path inside the zip file in this example is simply the name of the zipped application. You create a *System.IO.MemoryStream* object *zipFile* to pass it to the ZipFile constructor as required by #ZipLib, so to it, the memory data looks like a usual filestream. Finally, the variable *uzdata* contains the unzipped data.

Once you've uncompressed the application data in memory, you need to activate it. The starter application creates an *Assembly* from the *uzdata* byte array, which is invoked by activating its entry point:

```
Assembly assembly =
                  Assembly.Load(uzdata);
assembly.EntryPoint.Invoke
                  (null, new object[]{args});
```

I used *null* as *Invoke*'s first argument because the entry point, which corresponds to the *Main()* method of a C# application, is a static method. As the second argument, I pass the arguments passed to the *void Main(string[] args)* method of the starter application. This trick lets you transparently pass any command-line argument passed to the starter to the packed application. Consequently, when app.exe is started, it appears that the arguments come directly from the command line. You have to pack the argument as an object array to pass them to the *Invoke* method.

Alternatively, you can rely on reflection code to find the types in the assembly and invoke methods on them. This is useful when app.exe doesn't have an entry point, or when you want to invoke other methods. The startup time is usually smaller than starting app.exe directly because of lower disk overhead.

To make the code work, you must compile the starter application like this:

```
csc /t:winexe /out:starter.exe starter.cs
                              AssemblyInfo.cs
                  /r:zip.dll /res:app.resources
                              /win32icon:App.ico
```

Here, you create a Windows executable. Because I wanted to preserve version information of the original file, I used two additional files—AssemblyInfo.cs and App.ico—which come from the original app.exe. If you have the app.exe source code, you can reuse them; otherwise, you have to write some .NET reflection code to extract the assembly information from app.exe and generate AssemblyInfo.cs in Visual Studio format from it. You can also extract the icon file from the app.exe. .NETZ already contains code to do this automatically. It also compiles the starter application programmatically using the *System.CodeDom.Compiler.ICodeCompiler* interface with *Microsoft.CSharp.CSharpCodeProvider*.

You can rename starter.exe back to app.exe later if you like. This way, you distribute starter.exe and zip.dll, which are both smaller in size than app.exe alone.

Handling .NET DLL Files

If the sample app.exe depends on other DLLs, you normally don't need to do anything. At times, however, you may need to also zip the DLL files. Again, the technique I describe here does not work with DLL files placed in GAC, or is shared by more than one application not aware of the technique.

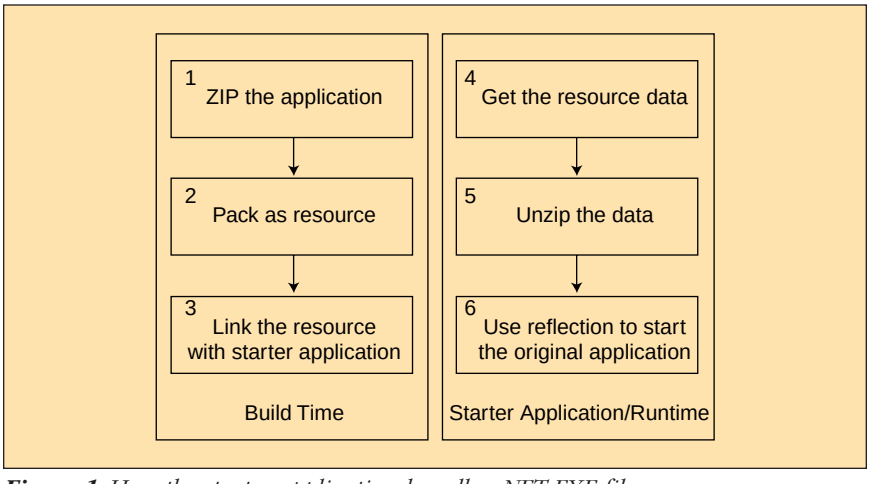


Figure 1: *How the starter application handles .NET EXE files.*

Suppose lib.dll is a DLL required by the app.exe you want to zip. First you would link app.exe with the normal unzipped version of lib.dll as you normally do. .NET has a built-in mechanism for resolving types and assemblies. When it fails, however, you can provide .NET with an assembly. This functionality is exposed by a hook in the *System.AppDomain* class. In .NET, every application executes in an application domain.

You need to handle this event:

```
AppDomain currentDomain =
                AppDomain.CurrentDomain;
currentDomain.AssemblyResolve += new
ResolveEventHandler
                (MyResolveEventHandler);
```

This code needs to be placed into the *Main* method of the starter application. The trick for this event to be activated is to place the app.exe assembly activation code in another separate method that is called by the starter's *Main* method.

Once you zip the lib.dll into lib.zip, then you can also pack it as a resource file with the starter application, as you did with the app.zip. This is preferable if you want to have a single EXE file; otherwise, you can leave it as a separate file. However, you need to rename the file to something different from lib.dll, given that .NET looks for this name, and it appears like a corrupted file to .NET. You can leave the name lib.zip or you can be creative and rename "lib.zip" to "libz.dll." Alternatively, you can save the lib.zip data in a SQL database table and retrieve it from there.

Listing Two is code that activates the DLL in *MyResolveEventHandler*. For the sake of example, suppose that the zipped DLL is a file in the same directory as the starter application. The types in the DLL are resolved to the *AppDomain*. The .NETZ tool supports both DLLs packed as resources and as separate files.

Conclusion

I presented here a pure C# technique that uses reflection to compress the size of .NET executables that requires no native code. The method is straightforward to implement and offers lots of possibilities. I have combined all these steps in the .NETZ open-source tool that can be used like this: *netz [-b] [-c] [-s] [exe file] [dll files] [-i win32icon],* where *-b* (batch mode) generates a batch file and source; *-c* is the console exe, the default is *winexe*; *-s* (single exe) packs DLLs as resources; and *-i win32icon* is an optional icon file.

DDJ

Listing One

```
FileStream fs = new FileStream("app.zip", FileMode.Open, FileAccess.Read);
byte[] data = new byte[fs.Length];
fs.Read(data, 0, data.Length);
fs.Close();

ResourceWriter rm = new ResourceWriter("app.resources");
rm.AddResource("app.exe", data);
rm.Close();
```

Listing Two

```
public static Assembly MyResolveEventHandler(object sender,
                                             ResolveEventArgs args)
{
    int i = args.Name.IndexOf(',');
    string dllName = args.Name.Substring(0, i);

    // dllName equals "lib" in the example; mapped to the zipped filename
    dllName += "z.dll";

    // read the file and unzip the data as above code omitted ...
    byte[] uzdata = ...

    return Assembly.Load(uzdata);
}
```

DDJ

64-Bit Computing & DSPs

Writing efficient code for resource-constrained platforms

SHEHRZAD QURESHI

Writing efficient code for memory- and resource-constrained embedded platforms is difficult, and the entire process is compounded when dealing with data-intensive computations such as signal- and image-processing algorithms. Digital Signal Processors (DSPs) are less general-purpose than familiar desktop CPU architectures—IA32, PowerPC, AMD, and the like. However, DSPs are architected to excel at the computations typically found in signal- and image-processing applications. In this article, I examine how 64-bit architectural features of the Texas Instruments C6416, a fixed-point member of the TMS320C6000 (C6x) family of DSPs, engender significant performance boosts in common operations used in image processing. The C6416 is an updated and faster version of the older C62x fixed-point DSPs, and the C6x family also includes the floating-point C67x series. Specifically, I illustrate how packed data-processing optimizations that take advantage of double-word-wide memory accesses can be incorporated into C code using Code Composer Studio compiler intrinsics to enhance the performance of critical loops. Code Composer Studio (CCStudio) is TI's flagship IDE and in many respects looks and feels like Microsoft's Visual Studio. I've compiled all of the code accompanying this article using Version 2.20 of CCStudio, and tested it on a C6416 DSK (DSP Starter Kit).

My approach to DSP code optimization involves a series of stages:

1. Development of a MATLAB prototype to gain an in-depth understanding of the algorithm and underlying mathematics at a high level.
2. Implement a straightforward reference C/C++ implementation of the algorithm in Visual Studio. This is usually a MATLAB-to-C/C++ port that generates the same output as Step 1. Because the C6416 is a fixed-point processor, I generally convert any algorithms from floating point to fixed point here (see "Fixed-Point Arithmetic for Embedded Systems," by Jean Labrosse, *C/C++ Users Journal*, February 1998, for a discussion of fixed-point representations of numbers).
3. Get the code running on the DSP platform. If performance is unsatisfactory, begin the optimization process through techniques such as reducing the memory footprint and packed data-processing techniques via specialized compiler intrinsics.
4. After identifying bottlenecks, write gating loops in so-called "linear assembly" or hand-optimized native C6x assembly.

Obviously, you must get to at least Step 3 during development of a DSP-based embedded system. It is essential to profile the code at this point to garner whether the processing meets the stated time requirements (hopefully such requirements are available). After all, if the code is fast enough as is, there is not much to be gained from proceeding to Steps 3 or 4, save for your own personal edification. Experimentation with the compiler and understanding its capabilities must be stressed—it is easier to fiddle with the compiler optimizations than it is to code in assembly. The TI compiler has a "compiler directed feedback" feature that emits information on utilization of the processor's functional units, software pipelining, and other useful information that may point one towards which compiler optimizations may make sense in this context (see "Code Efficiency & Compiler-Directed Feedback," by Jackie Brenner and Markus Levy, *DDJ*, December 2003, for additional information).

Using Compiler Intrinsics

Intrinsics provide you with access to the hardware while letting you remain within the friendly confines of the C programming environment. They directly translate into assembly instructions for processor-specific features that standard ANSI C cannot support, and are inlined so that there is no function call overhead. A simple example is the *_abs* intrinsic. The C6x instruction set includes an instruction that computes the absolute value of a register in a single clock cycle,

> "Intrinsics provide you with access to the hardware while letting you remain within the friendly confines of the C programming environment"

which is going to be more efficient than a multicycle C ternary statement that includes a branch statement; for instance:

```
int absolute_value = (a>0) ? a : -a;
```

In CCStudio, this C statement can be replaced by:

```
int absolute_value = _abs(a);
```

All C6x intrinsics begin with a leading underscore. Another basic operation encountered in image-processing algorithms is saturated arithmetic, whereby the result of an operation of two pixels is clamped to the maximum or minimum value of a predefined range. Rather than code a multicycle series of C statements that implement the saturated add, you should use the *_sadd* intrinsic (see Examples 2–6 in *TMS320C6000 Programmer's Guide*, literature number SPRU198g). Some of the C6x instructions are quite specialized, and many (not all) C6x assembly instructions have intrinsic equivalents—a full list is enumerated in the programmer's guide to the C6x. For example, on the floating-point C67x DSPs, there are instructions

Shehrzad is an engineer at Labcyte Inc. Portions of this article were adapted from his forthcoming book Embedded Image Processing on the C6000 DSP *(Springer, 2005). He can be contacted at shehrzad_q@hotmail.com.*

(continued from page 78)
for single- and double-precision reciprocal approximations—*RCPSP* and *RCPDP*, respectively. These instructions, and their corresponding compilers intrinsics *(_rcpdp* and *_rcpsp)* can be used to either seed a Newton-Rhapson iterative procedure to increase the accuracy of the reciprocal or perhaps as the reciprocal itself, as accuracy requirements warrant. However, my focus here is not on the use of intrinsics for specialized operations, but rather on using intrinsics within image-processing loops for vectorizing code to use word-wide (32-bit), or preferably, double word-wide (64-bit) optimizations that operate on packed data.

Packed Data Processing

Image-processing algorithms are notorious memory hogs, as it takes large amounts of memory to store image pixels. While this may not be of huge concern to nonembedded developers who have oodles of available RAM, embedded developers do not have the luxury of being so cavalier. Hence, the first order of business in the optimization of image-processing algorithms is to make maximum utilization of fast on-chip memory. This often entails splitting the image into pieces and paging these pieces from external memory to on-chip RAM, where there is a reduced memory-access latency.

Once the processing completes, the next order of business is to go the other way—page the processed pixels back out to external memory. Equally salient to image processing and signal processing in general is this idea of "packed data processing," where the same instruction applies the identical operation on all elements in a data stream. This general concept is known as SIMD (Single Instruction, Multiple Data) processing in computer architecture circles, and is powerful because, more often than not, the operations acting on the data stream are independent of one another, leading to code-generation techniques that exploit this inherent parallelism. With the right optimization, code size is reduced and processor throughput fully maximized. As you might imagine, Texas Instruments is not the only chip developer to incorporate such techniques into its processor cores. Intel brought to market similar SIMD IA-32 instruction set extensions with MMX, SSE, SSE2, and SSE3, as did AMD with 3DNow!. In a nutshell, packed data processing boils down to storing multiple elements in a single register, then using specialized processor instructions to operate on this data. Here, I use compiler intrinsics to access these processor instructions from within C code.

For example, consider the sum of products between two vectors in Example 1, which is critical in signal processing. This operation appears in various forms, a prominent example being the convolution of two sequences. Suppose *h* and *x* are 16-bit integer quantities, perhaps representing fixed-point numbers in Q15 format. Then a high-level description of what the processor is doing within a loop kernel that implements this vector product sum would look like Example 2. Since C6x registers are 32 bits wide, by reading the data in 16-bit (half-word) chunks at a time, you waste half of the storage capacity of registers 1 and 2. By packing the data to store multiple elements of the stream within registers 1 and 2, you can reduce the load pressure on this loop, as illustrated in the pseudocode in Example 3.

$$y[i]=\sum_{i=0}^{N-1}(h[i])(x[i])$$

Example 1: *Sum of products of two vectors.*

```
for each i

   1. Load h[i] from memory address &h+i, and place in register 1.
   2. Load x[i] from memory address &x+i, and place in register 2.
   3. Multiply contents of register 1 and register 2, placing result
     in register 3.
   4. Add contents of register 3 to running sum stored in register 4.

end
```

Example 2: *Loop that implements the vector dot product.*

```
for i = 0 ... N-1 in steps of 2
  1. Load h[i] and h[i+1], starting from memory address &h+i, placing h[i] in the lower-half of register 1
       and h[i+1] in the upper-half of register 1.
  2. Load x[i] and x[i+1], starting from memory address &x+i, placing x[i] in the lower-half of register 2
       and x[i+1] in the upper-half of register 2.
  3. Multiply lower-half of register 1 by lower-half of register 2 and place 32-bit result in register 3.
  4. Multiply upper-half of register 1 by upper-half of register 2 and place 32-bit result in register 4.
  5. Add contents of register 3 to running sum stored in register 5.
  6. Add contents of register 4 to running sum stored in register 5.

end
```

Example 3: *Vector dot product using packed data.*

Actually, Example 3 is an embodiment of two optimizations that go hand-in-hand—packed data and loop unrolling. You alleviate the load pressure in the loop kernel by reducing the number of instructions using word-wide data access; for example, replacing what would be *LDH* (Load Half-Word) instructions in the first loop with *LDW* (Load Word) instructions in the second loop, and subsequently packing two 16-bit quantities into the 32-bit registers. The same optimization holds (and is even more advantageous) if data elements in the stream are 8-bit quantities; then, using the just mentioned example, each load would place four data elements in each register and operate on them accordingly. This strategy replaces four *LDB* (Load Byte) instructions with a single *LDW* instruction. Of course, specifying these sorts of loads and stores is not feasible in strict and portable ANSI C, and it is a risky proposition to completely rely on the compiler's code optimization engine to generate code that takes full advantage of SIMD instructions operating on packed data. This is where intrinsics come in.

All C6x DSPs have instructions and corresponding intrinsics that let you operate on 16-bit data stored in the high and low parts of a 32-bit register, as illustrated in Example 2. The C64x and C67x offer double word-wide access via the *LDDW* (Load Double Word) and *STDW* (Store Double Word) instructions and corresponding intrinsics. In this case, 64 bits worth of data are read into the register file, with elements packed into a pair of registers. A requirement is that arrays must now be aligned on certain boundaries to use these instructions. Arrays have to be word-aligned to use *LDW/STW*, and aligned on a double-word boundary to use *LDDW/STDW*. The C64x DSP builds upon this architectural feature by allowing nonaligned accesses of memory via various instructions such as *LDNW/STNW* (load/store nonaligned word) and *LDNDW/STNDW* (load/store nonaligned double word), which correspond to the intrinsics *_mem4* and *_memd8*, respectively. These nonaligned instructions constitute a significant advantage, especially in certain image- and video-processing scenarios when algorithms march along in 8-bit pixel (byte) boundaries. Without such instructions, you are locked out of the packed data optimization due to restrictions imposed by 32-bit or 64-bit alignment boundaries.

One of the simplest examples of using intrinsics to help guide the compiler to take advantage of packed data is the implementation of a function that zeros out an array. This function is similar to the Standard C Library *memset* function; see

Listing One. This function only works with arrays aligned on a double-word boundary, where count is a multiple of 8 and greater than or equal to 32. However, given these restrictions, this function offers advantages over the general-purpose *memset* function, which by necessity must be more conservative than *memclear* to maintain generality (see Listing Two). The *_nassert* intrinsic in Listing One is an example of an intrinsic that does not generate any code, rather it asserts to the compiler that the address of the pointer passed into the function is divisible by 8; in other words, aligned on a double-word boundary. To declare an aligned array use the *DATA_ALIGN* pragma (see Listing Three). The *MUST_ITERATE* pragma directive is a means of conveying information to the compiler regarding the number of times a loop iterates, commonly referred to as the "loop trip count." Through this directive, you can specify the exact number of times a loop executes, if the trip count is a multiple of some number, the minimum number of iterations through the loop, and so on. This pragma should be used wherever possible—especially when the minimum trip count is known because this information lets the compiler be more aggressive when applying loop transformations. The form of the *MUST_ITERATE* pragma used in *memclear* specifies that the loop is guaranteed to execute at least 32 times, and armed with this information the compiler can proceed to unroll the loop. Loop unrolling is a code optimization technique where the kernel of a loop is expanded by some factor X—and the loop stopping condition adjusted to N/X—with the intent of reducing the number of branches. By reducing the branch overhead, the efficiency of the loop is increased, and it also allows for better scheduling of instructions contained within the loop kernel.

By stipulating the minimum number of iterations through the *memclear* loop, the input pointer casted to a long (64-bit) type, and guaranteeing alignment of *lptr* to a 64-bit boundary via *_nassert*, the compiler is given numerous pieces of information so that it can generate a loop that runs faster than an equivalent *memset*-like function. The compiler is now free to use the *LDDW/STDW* (load/store aligned double word) instructions to initialize the 64-bit number pointed to by *lptr*. The more conservative code compiles to assembly language using *LDB/STB* (load/store byte) instructions to initialize the 8-bit values pointed to by *ptr*, and a series of these instructions is not as efficient as a series of *LDDW/STDW* instructions due to the lessened throughput of the data flowing through the DSP. Figure 1 highlights the difference in operation between *memclear* and *memset* (as defined in Listing One). In a conservative, but more general, implementation, successive *LDB/STB* instructions are used for accessing and storing array elements. In a single iteration of *memclear*, each *LDDW* instruction loads 64 bits of data into a register pair. The assembly code would use two *MVK* (move constant) instructions to zero out both registers each time through the loop, then *STDW* to send the packed data back into the storage array pointed to by *lptr*.

Optimization of the Center-of-Mass Calculation

The "isodata" clustering algorithm for automatic threshold detection (see the text box "Autonomous Threshold Detection") is used in image segmentation. This algorithm calls for computing the center-of-mass of two portions of an image histogram, where the histogram is split by the current threshold. The isodata algorithm also entails repeatedly performing these center-of-mass computations, as the iterative procedure continues until a "good" threshold value is found. The center-of-mass calculation is illustrated graphically in Figure 2. The threshold *T=150* bifurcates the histogram, with c_1 the center-of-mass of the left portion of histogram and c_2 the center-of-mass to the right of *T*. The mechanics of this algorithm can be transformed so that you can employ the packed data-processing optimization via intrinsic functions to improve the performance of the algorithm. Furthermore, this optimization is an example where the nonaligned double-word instructions present only in the C64x series lets you use 64-bit memory accesses where you otherwise would not be able to.

A description of the procedure for computing the center-of-mass of a region of the histogram is simple enough. Referring to Figure 2, computing the center-of-mass to the left of *T* requires multiplying each histogram count by the bin number, summing those results together, and then dividing by the sum of the counts from 0 to *T*. In mathematical terminology, this translates to the equation in Example 4. Computation of the center-of-mass to the right of *T* is the same, except that the limits of summation change accordingly. Focusing attention on the numerator of this expression, note that you could state the numerator in terms of a dot product; in other words, the sum of products of two vectors, the values of the actual histogram, and the bins for that portion of the histogram we are summing over. So instead of loops like those in Listing Four, you can rewrite them and replace the variable *ii* in the loop kernels with an array indexing operation. Listing Five shows this modification, where the array *pixval* (consisting of an integer ramp) is used in lieu of *ii*. You can now vectorize the operation because you are willing to sacrifice memory usage (the *pixval* array for a 16-bit image would be large and would most likely preclude this optimization). If both the *hist* and *pixval* arrays are aligned on a double-word boundary, you might consider replacing half-word accesses with double-word accesses, reading and writing four elements at a time, and packing two elements per register. If you can make the assumption that the starting value of *ii* is divisible by 4 and *T-ii* is also divisible by

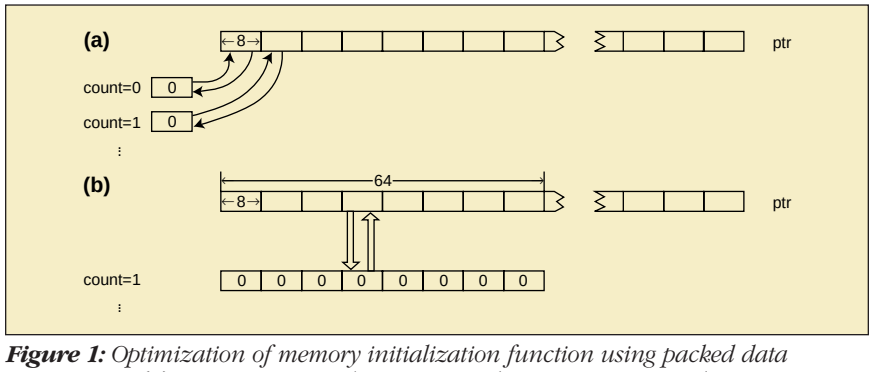


Figure 1: *Optimization of memory initialization function using packed data instructions. (a) Conservative implementation where successive* LDB/STB *instructions are used for accessing and storing array elements. (b) One iteration illustrating how* LDDW, MVK, *and* STDW *instructions are used to zero out 64 bits worth of data in one fell swoop.*

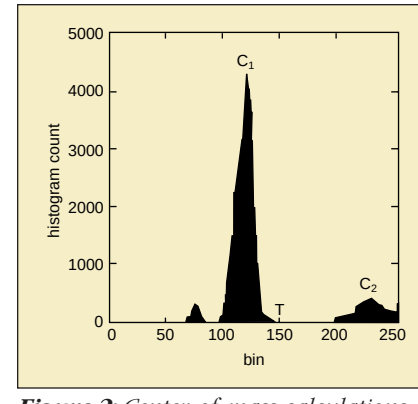


Figure 2: *Center-of-mass calculations in the isodata algorithm. The function depicted is an image histogram.*

$$\text{center-of mass}_{0\rightarrow T} = \frac{\sum_{i=0}^{T}(i)(hist[i])}{\sum_{i=0}^{T} hist[i]}$$

Example 4: *Center-of-mass calculation used in isodata threshold detection.*

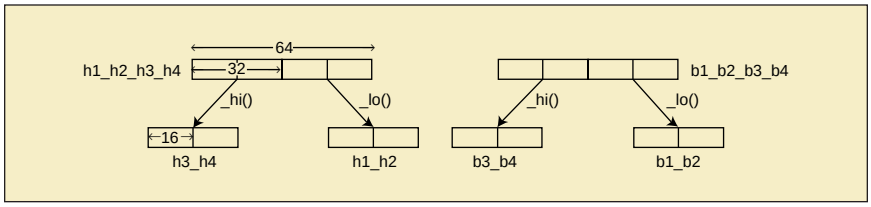


Figure 3: *Use of the* _lo *and* _hi *intrinsics to split 64-bit elements into 32-bit elements. 64-bit data elements are read into memory using the* _memd8_const *intrinsic.*

4 (64 bits divided by 16-bit elements), which is equivalent to saying the number of iterations through the loop is divisible by 4, then the loops in Listing Six would suffice on both the C67x and c64x DSPs.

The form of the *MUST_ITERATE* pragma in Listing Six tells the compiler that the number of iterations is divisible by 4, and this in conjunction with *_nassert* should be a sufficient trigger for the compiler to apply packed data optimizations using *LDDW* and *STDW*. Unfortunately, you can make no such assumption about the number of times through either loop and by extension can make no such claim about the starting value of *ii* in the second loop. This fact prevents double word-wide accesses on the C67x; however, with the C64x, you can take advantage of the nonaligned double-word instructions *LDNDW* and *STNDW* to improve the loop performance. Listing Seven shows the contents of the *dotprod* function that is used to compute the numerator of Example 4, assuming the existence of a global array *hist*. The use of *double* in the declarations for *h1_h2_h3_h4* and *b1_b2_b3_b4* may seem odd at first, but the type *double* in this context is just used as a placeholder to signify 64 bits of storage. The loop has been unrolled by a factor of 4 and the *_memd8_const* intrinsic, which generates code that uses *LDNDW*, is used to read 64 bits, or two words worth of data, into *h1_h2_h3_h4* and *b1_b2_b3_b4*. Next, both of these 64-bit elements, which in reality are stored in register pairs, are "split" into 32-bit upper and lower halves via the *_lo* and *_hi* intrinsics. At this point, you are left with a situation like that in Figure 3, with four 32-bit elements.

Although you could split the contents of the four registers again, you need not resort to that course of action. There are assembly instructions available that multiply the 16 LSBs (least significant bits) of one register by the 16 LSBs of another register. Similarly, there are assembly instructions for multiplying the 16 MSBs (most significant bits) of one register by the 16 MSBs of another register. There are actually four different variants of each of these instructions, pertaining to whether the data contained within the portion of the register are to be treated as signed or not, and whether the result should be truncated down to 16 bits or returned as a 32-bit entity. In general, multiplication of two 16-bit quantities yields a 32-bit result. As the comments in Listing 7 indicate, the *_mpyu* (16 unsigned LSBs multiplied together and returned as 32-bit quantity) and *_mpyhu* (16 unsigned MSBs multiplied together and returned as 32-bit quantity) intrinsics are used to perform four packed-data multiplications. These quantities are then added to the accumulators, four of which are used to avoid loop-carried dependencies that inhibit parallelism.

A loop "epilogue" of sorts is required to clean up. Because you can make no claims about loop count being divisible by 4, you need to wrap up the dot product computation using a final "traditional" C loop, which iterates at most three times. Alternatively, you could zero pad the arrays such that you are guaranteed to iterate a number of times divisible by 4. We have now succeeded in fully vectorizing the loop kernel, given the memory bandwidth of the architecture, but in fact have yet to reach the denouement of this story. A hallmark of DSP architectures is the fabled multiply-and-accumulate, or MAC, instruction, and it is not surprising

Autonomous Threshold Detection

Broadly speaking, image segmentation is the task of isolating those parts of images that constitute objects or areas of interest and separating these objects or areas from the image. Once this separation has been achieved, various characteristics (center-of-mass or area, for instance) can be computed and used toward a particular application. One such means of accomplishing image segmentation is to threshold the image, whereby all pixels less than some threshold *T* are set to zero while the remaining pixels are left alone. There are different twists on this idea—for example, setting all pixels greater than *T* to 0 and the remaining pixels set to some other value *V*—but the basic concept is the same. A central problem facing this scheme is how to choose a suitable value for *T*. If the input image is guaranteed to be of high contrast, then simply selecting a brightness threshold somewhere within the middle of the dynamic range may be sufficient (128 for 8-bit images). Obviously, this is not always going to be the case and there are a variety of algorithms that attempt to derive a "good" threshold from the histogram of the image—where the goodness criterion is one where the number of falsely classified pixels is kept to a minimum. One such means is to use known properties of the image to select the threshold value. For example, in the case of optical character recognition (OCR) applications, it may be known that text covers 1/p of the total canvas area. Thus, it follows that the optimal algorithm for OCR is to select a threshold value such that 1/p of the image area has pixel intensities less than some threshold *T* (assuming the text is dark and the sheet is white), which is easily determined through inspection of the histogram. This method is known as "p-tile-thresholding."

Alternative techniques relying on the morphology of the histogram are used where such knowledge is not available. One such method is the iterative isodata clustering algorithm of Ridler and Calvard (see "Picture thresholding using an iterative selection method," *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8:630-632, 1978). The image histogram is initially segmented into two sections starting with an initial threshold value T^0 such as 2^{bpp-1}, or half the dynamic range. The algorithm then proceeds as follows:

1. Compute sample mean of the pixel intensities of the foreground mf.
2. Compute sample mean of the pixel intensities of the background mb.
3. Set $T^{i+1} = (m_f + m_b) / 2$.
4. Terminate iteration if $T^{i+1} = T^i$, else go to 1.

This method has been shown to work well under a variety of image contrast conditions.

—S.Q.

that the C64x provides just the intrinsic you need here to cut down on the number of operations within this loop kernel. TI provides another set of intrinsics that map to so-called "macro" instructions, and one of these is DOTP2. This instruction, accessed in C via *_dotp2*, performs a MAC-like operation on packed 16-bit data; that is, it returns the dot product between two pairs of packed 16-bit values, as in Figure 4 for one of the *_dotp2* "invocations" (remember, while they may look like function calls they are in reality inlined functions). With this final tweak, the number of add operations is further reduced, as *_dotp2* subsumes two of the four adds present in Listing Seven. Listing Eight is the fully optimized dot product function that can be used within an isodata image segmentation implementation.

Conclusion

While the aligned word (LDW/STW) and double-word (LDDW/STDW) wide instructions are useful in the application of packed data processing code optimization techniques, the C64x line of DSPs augments this functionality with unaligned variants of these instructions. There are many image- and video-processing algorithms that call for marching across image dimensions in step sizes that are not even multiples of 4 or 8 bytes. Consequently, without these unaligned instructions, you're locked out of utilizing 64-bit computation methods. Two good papers on C64x-specific code optimization strategies as they pertain to digital-image processing are "VLIW SIMD Architecture Based Implementation of a Multi-Level Dot Diffusion Algorithm" by Ju and Song (sips03.snu.ac.kr/pdf/adv_prog.pdf) and "Implementation of a Digital Copier Using TMS320-C6414 VLIW DSP Processor" by Hwang and Sung (mpeg.snu.ac.kr/pub/conf/c61.pdf). For a more detailed exposition on C6x code optimizations, refer to the TI documentation (http://www.ti.com/) or "Preferred Strategies for Optimizing Convolution on VLIW DSP Architectures" by Sankaran, Pervin, and Cantrell (http://www.crest.gatech.edu/conferences/cases2004/paper/sankaran.pdf).

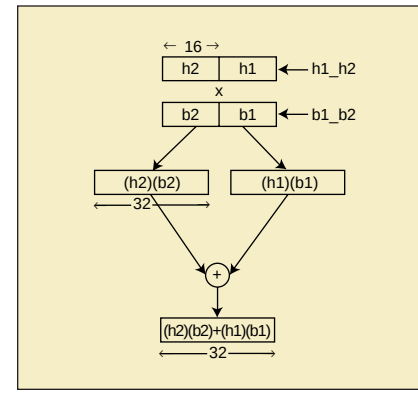


Figure 4: *Usage of the* _dotp2 *intrinsic in Listing Eight. The same operation is performed on* h3_h4 *and* b3_b4.

DDJ

Listing One

```
void memclear( void * ptr, int count )
{
  long *lptr = ptr;
  _nassert((int)ptr%8==0);
  #pragma MUST_ITERATE (32);
  for (count>>=3; count>0; count--)
    *lptr++ = 0;
}
```

Listing Two

```
void memset( void *ptr, int x, int count ) {
    char *uch = ptr;
    for (; count>0; count--) *uch++ = x;
}
```

Listing Three

```
#pragma DATA_ALIGN(double_word_aligned_array, 8)
unsigned char double_word_aligned_array[256];

#pragma DATA_ALIGN(word_aligned_array, 4)
unsigned char word_aligned_array[256];
```

Listing Four

```
/* "left" center-of-mass numerator */
for (ii=0; ii<T; ii++)
  sumofprod1 += ii*hist[ii];
/* "right" center-of-mass numerator */
for (ii=T+1; ii<MP; ii++) /* MP=255 for 8-bit images */
  sumofprod2 += ii*hist[ii];
```

Listing Five

```
/* pixval = {0, 1, 2, ..., 255} for an 8-bit image */

for (ii=0; ii<T; ii++) /* left */
  sumofprod1 += pixval[ii]*hist[ii];
for (ii=T+1; ii<MP; ii++)
  sumofprod2 += pixval[ii]*hist[ii];
```

Listing Six

```
#pragma MUST_ITERATE(,,4)
_nassert((int)pixval%8 == 0)
_nassert((int)hist%8 == 0)
for (ii=0; ii<T; ii++) /* left */
  sumofprod1 += pixval[ii]*hist[ii];
#pragma MUST_ITERATE(,,4)
_nassert((int)pixval%8 == 0)
_nassert((int)hist%8 == 0)
for (ii=T+1; ii<MP; ii++) /* right */
  sumofprod2 += pixval[ii]*hist[ii];
```

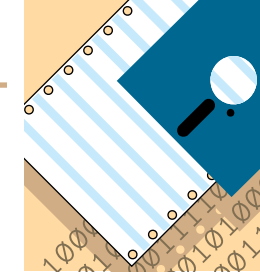
Listing Seven

```
unsigned long dotproduct(int lo, int hi)
{
  /* 0, 1, 2, ..., 255 */
  static const unsigned short pixval[] =
                                {0,1,2, /* 3,5,...,252 */ ,253,254,255};
  unsigned long sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0, sum;
  const int N = hi-lo;
  int ii=0, jj=lo, remaining;
  double h1_h2_h3_h4, b1_b2_b3_b4;
  unsigned int h1_h2, h3_h4, b1_b2, b3_b4;

  /* unrolled dot-product loop with non-aligned double word reads */
  for (; ii<N; ii+=4, jj+=4)
  {
    h1_h2_h3_h4 = _memd8_const(&hist[ii]);
    h1_h2 = _lo(h1_h2_h3_h4);
    h3_h4 = _hi(h1_h2_h3_h4);

    b1_b2_b3_b4 = _memd8_const(&pixval[ii]);
    b1_b2 = _lo(b1_b2_b3_b4);
    b3_b4 = _hi(b1_b2_b3_b4);

    sum1 += _mpyu(h1_h2, b1_b2); /* (h1)(b1) */
    sum2 += _mpyhu(h1_h2, b1_b2); /* (h2)(b2) */
    sum3 += _mpyu(h3_h4, b3_b4); /* (h3)(b3) */
    sum4 += _mpyhu(h3_h4, b3_b4);  /* (h4)(b4) */
  }
  sum = sum1 + sum2 + sum3 + sum4;
  /* loop epilogue: if # iterations guaranteed to
   * be a multiple of 4, then this would not be required.
   */
  remaining = N - ii;
  jj = N - remaining;
  for (ii=jj; ii<N; ii++)
    sum += hist[ii]*pixval[ii];
  return sum;
}
```

Listing Eight

```
unsigned long dotproduct(int lo, int hi)
{
  /* 0, 1, 2, ..., 255 */
  static const unsigned short pixval[] =
            {0,1,2, /* 3,5,...,252 */, 253,254,255};
  unsigned long sum1 = 0, sum2 = 0, sum;
  const int N = hi-lo;
  int ii=0, jj=lo, remaining;
  double h1_h2_h3_h4, b1_b2_b3_b4;
  unsigned int h1_h2, h3_h4, b1_b2, b3_b4;

  /* unrolled dot-product loop with non-aligned double word reads */
  for (; ii<N; ii+=4, jj+=4)
  {
    h1_h2_h3_h4 = _memd8_const(&smoothed_hist[ii]);
    h1_h2 = _lo(h1_h2_h3_h4);
    h3_h4 = _hi(h1_h2_h3_h4);

    b1_b2_b3_b4 = _memd8_const(&pixval[ii]);
    b1_b2 = _lo(b1_b2_b3_b4);
    b3_b4 = _hi(b1_b2_b3_b4);

    sum1 += _dotp2(h1_h2, b1_b2); /* see Figure 4 */
    sum2 += _dotp2(h3_h4, b3_b4);
  }
  sum = sum1 + sum2;
  /* loop epilogue: if # iterations guaranteed to
   * be a multiple of 4, then this would not be required.
   */
  remaining = N - ii;
  jj = N - remaining;
  for (ii=jj; ii<N; ii++)
    sum += smoothed_hist[ii]*pixval[ii];

  return sum;
}
```

DDJ



The Blind Men and The Elephant

Michael Swaine

From one perspective, it makes no difference whether a programming problem is solved by 200 lines of Fortran code or a handful of Java classes or a set of Lisp functions. Any general-purpose programming language can, in principle, solve any problem that any other general-purpose programming language can solve: That's what it means to be a general-purpose programming language. And if it's solved, it's solved.

From another perspective—the working programmer's—it can make a great deal of difference what language you use to solve a problem. There is typically a domain of problems that is natural to a given programming language. Using the right tool saves time and effort.

Exploring the implications of a set of premises is what Prolog was designed for. Fortran was originally created to speed up large but straightforward mathematical calculations. Snobol is all about manipulating strings of characters. When Perl fanatics show you that they can build a spreadsheet in Perl, they may be demonstrating the flexibility of Perl or demonstrating their chops while busting yours, but they're probably not demonstrating sensible professional programming behavior.

What goes for programming languages also goes for other programming tools, whether they are called libraries or toolsets or frameworks or methodologies or whatever. Like the hammer that conditions the carpenter to think of every problem as a nail, programming tools are all just different paradigms, different perspectives from which we look at problems. All perspectives are in one sense equivalent, but for any given problem, one perspective may reveal a solution much more quickly and naturally than another.

I've been thinking a lot recently about this idea of perspectives that are in one sense equivalent but in another sense very different. I was led into these thoughts by two things: playing around with the latest version of Mathematica, and rereading the key chapter of Stephen Wolfram's *A New Kind of Science*. These thoughts, in turn, led me to research an old story about some blind men and an elephant, and to realize that the moral of that story might be very different from what I always took it to be.

All of which led to the following tentative reflections on paradigms and perspectives and programming and science and relativism.

A Hindu Parable

Do you know the story of the blind men and the elephant? If you do, you probably either have read the poem by John Godfrey Saxe or have been introduced to the blind men by some speaker or writer using the story to illustrate a point. The poem ties up the story with a straightforward moral, and the essayists and lecturers use it similarly, but the original Hindu parable, at least in the version that I've seen, is surprisingly ambiguous.

In the parable, a raja sent a servant to gather several men who were born blind and to have them examine an elephant and report on their findings. The servant showed each blind man a different part of the elephant, and predictably, each reported on the aspect of the elephant that he had experienced—the one that had touched its side saying that it resembled a wall, the one that felt its tusk saying that it resembled a plowshare, and so on. And each thought that his experience of the elephant was the complete and correct view of the beast. So certain were they that they came to blows over this matter of the nature of the elephant. The raja, according to the parable, was delighted with this scene. Go figure.

Actually, there is a frame-story wrapped around this one, in which the Buddha relates this story of the raja and the servant—who seems to enjoy playing practical jokes on the visually impaired—for a purpose: The Buddha wants to teach his disciples a lesson about those who argue over whether the world is infinite or finite or whether the soul dies with the body or lives forever. His lesson is that, in their quarreling, each clings to his own view and sees only one side of the issue.

I've heard or read the story several times, always presented to make a point about the need to recognize the limits of your present perspective. But on rereading it, it seems ambiguous. Does the Buddha expect his disciples to "see" the true nature of the universe, or merely to recognize their own blindness? The blind men are, after all, congenitally blind. Are we to believe that there is what Albert Einstein called, in a different context, a "privileged perspective," a nonblind view of the elephant, of reality? Or do we get only a choice of different but equivalent perspectives—which are not views of some underlying reality, but are themselves all there is to reality? I think the story can be read either way.

Blind Men and Programming Paradigms

In the case of programming languages as perspectives, it seems to me that the

Michael is editor-at-large for DDJ. *He can be contacted at mike@swaine.com.*

second interpretation of the story is the relevant one. In other words, there is no elephant: no privileged programming language, or privileged programming paradigm, merely a possibly infinite set of functionally equivalent ways of going about solving problems of computation.

Mathematica, the symbolic mathematics software invented by Stephen Wolfram, is a good playpen for fooling around with different programming paradigms. You can use it as a cookbook for procedural programming languages like Basic or Fortran:

```
z = a;
Do[Print[z *= z + 1], {i, 3}]
```

or as a functional language like Lisp, in which everything is a function call and functions can be treated as data objects:

```
Nestlist[(1 + #) ^ 2 &, x, 3]
```

or as a string-manipulation language like Snobol or some of the popular "little" or scripting languages:

```
StringReplace[s,
   {"AG" -> "AC", "GT" -> "GT"}]
```

or as a Prolog-like rule-based language:

```
p[x_ + y_] := p[x] + p[y]
p[a + b + c]
```

or define objects as in object-oriented programming languages, or mix paradigms in one program.

But there is, according to Wolfram, one unifying idea underlying Mathematica: Everything can be represented as a symbolic expression of the form

```
head[arg1,arg2…].
```

Every operation in Mathematica is ultimately a transformation of such a symbolic expression. So maybe for Mathematica, there is a privileged perspective: symbolic expressions transformed by transformation rules.

Does the fact that Mathematica seems to have a privileged paradigm mean that there is some privileged programming paradigm in a general sense? I don't think so. Surely Mathematica's privileged perspective is simply a consequence of its architecture.

But what about assembly language or machine language? Might that be the "true" perspective against which high-level languages are merely distorted views, not from blindness maybe, but through tinted glasses?

I suspect not. I think that when we talk about the perspective of a programming language, we are not talking about a particular implementation on particular hardware, but about the programming paradigm behind that language — object-oriented programming, for example, or declarative programming. And if the question is really about paradigms, and about full computational systems that include the hardware, then it doesn't seem that there is any privileged perspective. There are practical reasons for building the underlying logic hardware the way we do, but not fundamental logical reasons.

The CA paradigm

It is, of course, of great practical importance that different programming paradigms work better for different purposes. Particular paradigms are easier to apply, more natural in particular contexts.

> "Mathematica is a good playpen for fooling around with different programming paradigms"

Stephen Wolfram's preferred programming paradigm seems to consist of the following components:

1. A set of transformational rules.
2. Data to operate on.
3. An engine that applies the rules to the data.

That's loose enough to describe Mathematica or an expert-system inference engine or any of a number of other programming systems. If you add the assumptions that the data enter only at the beginning of the process as the initial condition of the system, and that the engine keeps applying the same rules to the output of its previous application of the rules, then what you have is a pretty good definition of a cellular automaton (CA). The most famous example of a cellular automaton is the Game of Life popularized in the 1970s in the pages of *Scientific American* magazine by John Horton Conway and Martin Gardner.

The CA paradigm turns out to be capable of emulating a Turing machine, and is therefore computationally equivalent to any general-purpose programming language. It's a paradigm that Wolfram has spent the past 20 years studying. The question raised by his research is: Is the CA paradigm the best perspective for studying the universe? Is it, in fact, the universe's privileged perspective?

My reading of Stephen Wolfram on science is that, contrary to the situation with programming, there is a privileged perspective in science. That may not seem particularly strange: It is hardly shocking, in fact may be a little old-fashioned, to suggest that there is a fundamental reality behind our various views of the universe, that there is a real elephant behind the differing reports of the blind men. The curious thing, though, is that, for Stephen Wolfram, this privileged perspective is itself functionally equivalent to a programming language.

Blind Men and the Universe

I've written before about Wolfram's magnum opus *A New Kind of Science*, but I never really did justice to the key chapter of the book, the one in which he explains his Principle of Computational Equivalence. I don't know that I can do better now. I keep trying to absorb it, but I begin to suspect that the simple writing style that Wolfram adopted for the book is inadequate for fully explaining this concept, which he claims is broader than previously established deep results about computation, with richer implications than the laws of thermodynamics: a new law of nature, an abstract fact, and a powerful and enlightening definition.

That's a lot to claim. But if you take Wolfram seriously, and his intellect makes it foolish not to at least give him a hearing, the concept is central to understanding a great many things, including the question of whether or not there is a privileged perspective on the universe. He says:

> [I]t has become particularly common in the academic humanities in the past few decades to believe that there can be no valid absolute conclusions about the world — only statements made relative to particular cultural contexts…But the Principle of Computational Equivalence implies that in the end essentially any method of perception and analysis that can actually be implemented in our universe must have a certain computational equivalence, and must therefore at least in some respects come to the same absolute conclusions.
>
> —Stephen Wolfram, *A New Kind of Science,* p. 1131

Before he can explain his Principle of Computational Equivalence, though, Wolfram has to demonstrate what he could call (but doesn't) the Principle of Computational Ubiquity.

Part of the 1200-page book consists of detailed demonstrations that computations that are similar to, and computationally equivalent to, cellular automata can be found just about everywhere in nature. Wolfram's researches take him into crystal structures, fracture patterns in materials, fluid flow, and patterns in biological morphology. He examines growth patterns in plants and animals, with hundreds of illustrations showing the similarity between the output of a simple program and the structure of a particular leaf. He reasons from the ubiquitous appearance of the angle 137.5 degrees in plant structures to the likelihood of an underlying process that is very much like a cellular automaton. His detailed study of the shapes of seashells is reminiscent of Darwin.

Other chapters in the book explore the way in which such seemingly simple computations show up in other realms, like fundamental physics. One highly interesting assumption of Wolfram's is that these discrete computations are adequate to capture all of physics. He doesn't insist, but he does apparently believe, that the universe is discrete, and that continuous functions are a mathematical abstraction with no direct realization in nature.

There are, I guess, two points to be made here. First, that Wolfram finds computations everywhere. Where the ancients thought that all was fire or earth or air or water or some combination of these elements, and more recently "all is atoms" was a mantra of science, Wolfram holds that "all is computation." And second, the computations that he finds everywhere tend to be, or at least appear to be, quite simple, either cellular automata or equivalent systems.

The reason for this, Wolfram tells us, is that there are no more complex calculations than these simple CAs.

PCE

Wolfram's Principle of Computational Equivalence, the punchline of his book, states that almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication. In particular, simple CA systems no more elaborate than Conway's Game of Life are computationally equivalent to powerful computer systems.

It says that once you get beyond very simple systems, all systems immediately attain the highest level of complexity possible, and are computationally equivalent to all other nonsimple systems. The Principle of Computational Equivalence, Wolfram says, "tells us what kinds of computations can and cannot happen in our universe [and] summarizes purely abstract deductions about possible computations, and provides foundations for more general definitions of the very concept of computation." [*A New Kind of Science*, p. 719.] It introduces a new law of nature asserting that "no system can ever carry out explicit computations that are more sophisticated than those carried out by systems like cellular automata and Turing machines."

One consequence of the Principle is that the detailed behavior of most systems that are not trivially simple cannot be known without in effect running the computation and observing the behavior directly. Because any accurate theory, model, or simulation of the system is necessarily of the same degree of complexity as the system itself. This runs counter to our idea of how science works, but this is, Wolfram says, because science today restricts itself to those systems that are simple enough to produce only repetitive or nested patterns of behavior. Science today ignores the vast majority of the processes of nature, looking only at those where easy answers can be found. Whereas the new kind of science revealed by Stephen Wolfram boldly takes on all the hard questions that no scientist has ever had the courage or imagination to tackle before.

Sorry; I got carried away. It's hard to characterize Stephen Wolfram's views without a little of the Wolfram ego slipping in.

So what is the scientific status of this Principle of Computational Equivalence? Well, the whole of *A New Kind of Science* is an argument for the Principle. And Wolfram acknowledges that the Principle is so fundamental that it may not be directly testable by the conventional methods of science. But he argues that the large amount of data presented in the book and the new perspective that the book opens up strongly support the Principle. Perhaps, he suggests, various aspects of the Principle will come to be accepted, until eventually the whole thing seems too obvious even to mention.

Time will tell.

The Privileged Perspective?

Wolfram titled his book *A New Kind of Science* because, essentially, nobody has ever done science in the way he proposes. Scientific method has traditionally consisted of looking at complex processes and discovering simple regularities in the output of these processes. These regularities are invariably either repetitions or, as in the case of fractals, nested regularities. Wolfram proposes studying the processes themselves in all their computationally irreducible complexity. Because he finds computational systems everywhere in nature, he concludes that this means studying the behavior and properties of computational systems that are equivalent to cellular automata.

And so, the image emerges of the entire universe as a vastly complex system creating itself anew each instant from a possibly simple set of initial conditions and a possibly simple transformational rule.

Now, although a CA can be emulated by a Turing machine or other programming paradigm, we know that one programming paradigm is usually the most convenient, the most natural, in a given context.

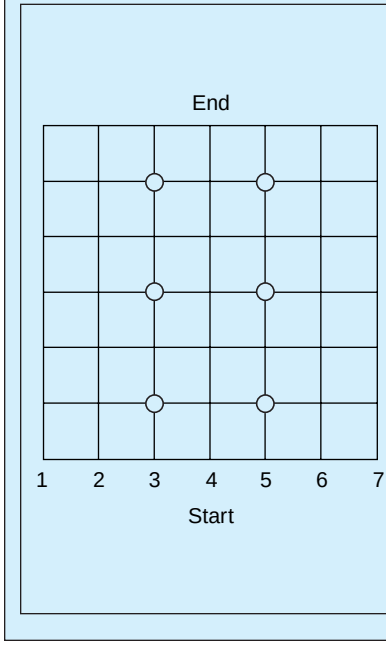
For the universe, is that most natural paradigm the cellular automaton?

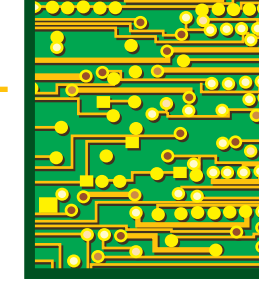
DDJ

Dr. Ecco Solution

Solution to "Dig That!," *DDJ*, February 2005.

1. Six probes; see Figure. Because you know the direction of the roads as they leave an intersection, each pair on each row determines the fate of the pipe on that row and the next.
2. Six hours is the best, I think. You may be able to cut out early if a detour and a return to the central path has been detected, but this is not guaranteed.





Long-Time Memories

Ed Nisley

High-level languages use a fairly straightforward model of system memory: It's arbitrarily large, read-write, uniform, and fast. The hideous details of memory management generally reside in the operating-system kernel, which interfaces with whatever hardware features the CPU and its surrounding chipset might provide. Apart from applying the same number of allocate and free operations to a given block, programmers generally don't need to worry much about the exact type of memory they're using.

Embedded programmers, on the other hand, must know far more about those hideous details, if only because their programs must handle weird hardware. Seemingly simple memory chips sport command interfaces, write operations may be six orders of magnitude slower than reads, and the chips can return bad data even before they wear out from overuse. Get one detail wrong and your product will fail unpredictably. Oh, and that's before you even think of writing a hard real-time application.

Let's see how we got here, then examine the sort of Flash memory one might find in a typical gizmo these days. You'll see how we've been spoiled by newfangled semiconductor memory. Perhaps the Bad Old Days are back?

Earliest Memories

ENIAC, one of the earliest digital computers, stored 1 bit in a pair of vacuum tubes wired as a set-reset flip-flop, a configuration called a "bistable multivibrator" in those days. This being a gadget designed by engineers, every flipflop displayed its bit on a neon bulb. Think of it: You could read the entire memory just by looking at the front panel!

Early programmers found ENIAC's 20 "accumulators" somewhat confining and, in 1953, Burroughs spliced on 100 words of magnetic core memory. That still wasn't nearly big enough, it was far too slow, and its three-phase power supply seemed excessive. More research was certainly needed.

Even before ENIAC, Atanasoff concocted a rotating-drum memory based on 1600 discrete capacitors. Pictures show spine-like contacts sticking out of the drum, so this thing obviously had serious reliability issues.

Replacing those capacitors and contacts with a smooth magnetic surface and a row of read-write heads boosted drum capacity to a few megabits, but the access time remained painfully slow, even by mid-1940's standards. Programmers learned to use the latency of the drum's rotation for I/O timing and compute-bound sections of code.

EDSAC, built in England around 1949, used what I think are the neatest storage devices ever made: mercury delay lines. A piezoelectric transducer launched acoustic pulses through pipes of liquid mercury to a receiving transducer. Analog feedback amplifiers regenerated the pulses on the fly to form a no-moving-parts, serial-access memory holding about 2 kilobytes of evanescent data. The access time, a few tens of milliseconds, was comparable to drum memory and far too slow for the ever-increasing speed of the rest of the system. Sound familiar?

Developed at roughly the same time as the delay lines, the Williams tube stored up to 2 kilobits in charged spots on the face of a cathode-ray tube, with the key advantage of electronic-speed random access to the bits. A pair of bottles held 128 40-bit words in the Manchester Mark I computer, the first machine to store both instructions and data in random-access memory. Yes, both in 128 words!

Magnetic core memory came back from a shaky start to sweep away all contenders in the early '60s. It had three key advantages:

- Fast access.
- Random addressing.
- Relatively low cost per bit, despite hand-threading all those teensy ferrite doughnuts.

Core was also nonvolatile, but most commercial systems didn't really take advantage of that fact, as programs and data were already far larger than any available memory.

Core remained the memory of choice for high-end systems until the late '70s, when integrated-circuit semiconductor memory finally became cheap and reliable enough. In fact, the Intel 1101 256-bit (!) static RAM chips were nothing more than ENIAC's bistable multivibrators: Half a dozen transistors scribed on a silicon chip replacing two hot triodes.

The neon indicators, alas, didn't fit.

Intel's 1702 2-kilobit EPROM, on the other hand, was almost unusably complex, requiring three power supplies (+5, +12, –12V), strobed –48V programming pulses on the data lines, and finicky UV erasing. But EPROM was nonvolatile and could actually form the basis of a recognizable embedded system, after microcontrollers got beyond the initial 4004 architecture.

In fact, small embedded systems through about the mid '90s typically featured a three-chip cluster: microcontroller, EPROM, and RAM. Smaller systems might omit the

*Ed's an EE, PE, and author in Poughkeepsie, NY. Contact him at ed.nisley@ieee.org with "*Dr Dobbs*" in the subject to avoid spam filters.*

RAM, tiny microcontrollers might have on-chip ROM, but the overall plan was about the same. In all cases, memory was pretty simple, as long as you remembered that you could read and write RAM, but only read EPROM.

Then came Flash memory.

Flash Flavors

NOR Flash, introduced by Intel in 1988, closely resembles EPROM, at least for read access. Each memory location, holding 8 or 16 bits depending on the chip, can be directly addressed and read in tens of nanoseconds, roughly the speed of large RAM chips.

NAND Flash, introduced by Toshiba in 1989, uses a serial interface that closely resembles a disk drive, if not a mercury delay line. The interface accepts commands, address, and data bits multiplexed over a few pins. Reading any particular location takes tens of microseconds, but reading successive memory addresses happens in a few tens of nanoseconds.

Unlike EPROM and the later EEPROM, Flash memory can be erased in relatively small sections. The CPU can store new data, thus enabling in-system updates and the miracle of in-flight program patching.

The names NOR and NAND vaguely suggest the internal structure of the memory arrays, but should indicate general categories: NOR means "direct access" and NAND means "serial access." As you might expect, there are myriad variations on the theme.

To inject some real-world numbers in the discussion, I'll use the Samsung KAB0xD100M-TxGP multichip memory. It has a 16-bit datapath (a "word") with access to 4M-word NOR Flash, 8M-word NAND Flash, and 2M-word pseudostatic RAM. The "x" characters are placeholders for digits that indicate various options and speeds that aren't relevant here. While you can find bigger and faster versions of each component, this one datasheet has all the pieces: http://tinyurl.com/3wxmt.

Most of the datasheet supplies the details that the hardware folks use to interface the chip with whatever microcontroller will be running the code. A few key specs, however, make life difficult for the software folks who might otherwise regard the Flash as just RAM with a really slow write cycle.

That's a really great way to kill a chip, if not an entire project.

Unlike traditional RAM or EPROM chips, Flash memory chips have both nonuniform addressing and a command structure. The address space includes control registers, data, room for metadata like ECC bits, hidden blocks of secret stuff, configuration settings, and so forth and so on. An on-chip state machine controls access to the chip's innards, so you must ensure that your program's model of that state machine either matches what it's actually doing or can force it into a known state. The datasheet gives the details, but expect to spend some time experimenting.

"Flash memory can be erased in relatively small sections"

NAND Flash

NAND Flash may be the easiest to understand, if only because its serial nature alerts you to something unusual. Each operation requires writing a command and address into the chip, then either writing or reading the appropriate data. The commands can reset the chip and read ID bytes and status, in addition to the expected data-read, -erase, and -write operations.

For example, reading a particular data word from the Samsung NOR Flash involves sending a *Read1* command followed by three address bytes specifying the page containing the word, all of which takes at least 120ns. The chip then transfers the entire page to an internal latch in a leisurely 10µs process, after which you can read all 256 words at a mere 50ns each and extract the particular word you wanted from that stream.

Obviously, NAND Flash is best used for applications that mimic a disk drive. Because it does not provide random access to words within a page, you cannot execute code from it or read widely scattered words with any alacrity.

After reading those 256 words, you must send another *Read1* command with the next page address and endure another 10µs startup delay. The net transfer rate is thus (10µs + 256×50ns)/256 = 90ns per word. That's actually not too shabby, as long as you're consuming data in a stream rather than sip by sip. If you're playing back audio or displaying a picture, it's the right hammer for the job.

Unfortunately, it's not quite that simple, because NAND Flash doesn't have nearly the same reliability as, say, the SDRAM in your server. Each 256-word page has a corresponding 8-word Spare Area for the ECC bits required to correct what's charmingly called "bit flip" in the main data. The ECC algorithm is up to you, but you should have one!

The Spare Area is accessible in two ways. You can read 256 data words, then continue to read the additional eight words, or you can use a *Read2* command to access just the Spare Area without slogging through the actual data.

Storing data works similarly, but you must erase an entire page before writing new words. Erasing a page requires up to 3ms (yes, 3000µs!), writing data takes (264×5ns), and the actual program operation takes up to 500µs, for about 3.6ms. You must compute and store the additional eight words in the Spare Area along with the main data, because there's no other way to write them.

In addition to bit flip, NAND Flash simply wears out with use. Permanent single-bit errors will occur after 1000 erase/program cycles on any given block. Single-bit ECC pushes the inevitable failure out to 100,000 cycles when you can expect a second permanent bit failure in a page. A transient bit flip in a page with a permanent error will cause an uncorrectable error, so you must factor that probability into your choice of ECC algorithm.

Even with ECC, the NAND Flash chip may report that an erase or programming operation has failed, in which case your code must relocate that page's data somewhere else. NAND Flash really does behave just like a disk drive: It has bad sectors and requires a mapping directory of some sort.

At least it's smaller than a drum memory and less toxic than a mercury delay line.

NOR Flash

Unlike NAND Flash, NOR Flash has a straightforward address and data interface: Present an address, assert the Read control line, and out pops the corresponding data.

It also has a control interface that accepts commands as data values written to specific addresses in a particular order. For example, to erase the whole chip, you write 0xaa to 0x555, 0x55 to 0x2aa, 0x80 to 0x555, 0xaa to 0x555, 0x55 to 0x2aa, and 0x10 to 0x555. Got that?

Now, if your first reaction is to figure the probability that a series of ordinary memory writes would accidentally trigger a chip erase, you've fallen into a classic embedded systems trap. Remember that the type of chip we're discussing was once known as Flash ROM:

Under normal circumstances, the chip never sees write operations because it's not RAM.

Pop Quiz: Compute the probability that such a sequence would arise at least once in a system writing to RAM. Hint: Your first guess is probably correct.

NOR Flash blocks must also be erased before being programmed, which the datasheet says takes 700ms (yes, 700,000µs) "typical." A block has 32K words, so the average value is 20µs per word. Programming a single word with data requires on the order of 10µs, making the overall bandwidth pretty dismal. The chip's data output contains various status flags during the programming operation, so your code can simply poll the chip to figure out when it's finished.

See the gotcha? If the memory produces status outputs instead of data, the code that should poll the outputs will crash when it tries to execute status bits fetched as instructions from different addresses in that same chip.

It turns out that early Flash chips had exactly that problem. Systems required either two Flash chips (one to run and one to program), executable RAM, the ability to run from the CPU's instruction cache, or some combination of tricks.

The Samsung part's NOR Flash is divided into two sections and can program or erase one while performing normal read operations from the other. A relatively small section, called the "Boot Block," generally holds the system's startup code, as well as the utility code required to reprogram the other, much larger, section with a new version of the main firmware.

The Samsung module also has a 2M-word RAM chip that can be used for instructions, but that's a feature of this particular multichip part rather than Flash in general.

Unlike NAND Flash, NOR flash is rated for over 100,000 updates to a single block before it fails, so ECC isn't absolutely required and there is no dedicated Spare Area for those bits. You'd be well advised, however, to compute an overall checksum for your data, as it's entirely possible for errors to creep in unannounced.

There is, however, a 32K-word Security Code area hidden in the Boot Block that can be programmed only once, then accessed only by a command sequence. That's where you put the unique Product ID codes that lock software to your gizmo, at least if you also believe in Santa Claus.

Reentry Checklist

I'll pick up the thread of Flash file systems later on. In the meantime, remember that any particular Flash memory chip will have more peculiarities than I've mentioned here. Expect the unexpected!

The History of Computing project is at http://www.thocp.net/index.htm. A 1946 paper on ENIAC at http://pages.cpsc.ucalgary.ca/~williams/509pdffiles/ENIAC.pdf gives an overview of how it all worked. See a decent picture of the replica Atanasoff-Berry Computer at http://perun.hscs.wmin.ac.uk/JHPC00/; the original ABC drum at http://www.csm.ornl.gov/ssi-expo/abcDrum.gif; and a Williams tube at http://www.cedmagic.com/history/williams-tube.html.

A history of nonmechanical storage, with some nice photos of core memory planes, is at http://www.allaboutcircuits.com/vol_4/chpt_15/4.html and an analysis of core memory development is at http://theory.lcs.mit.edu/classes/6.972/Core%20Report.html. A bit of core versus IC history is at http://www.eetimes.com/special/special_issues/millennium/milestones/whittier.html.

The Story of Mel shows how a Real Programmer puts a bizarre instruction set and drum memory to good use: http://www.jargon.net/jargonfile/t/TheStoryofMel.html.

The datasheet for Samsung's multichip NOR- and NAND-Flash plus RAM module is at http://tinyurl.com/3wxmt/, which encodes a nasty jawbreaker of a Samsung URL. If that doesn't work, start at http://www.samsung.com/, search for KAB01D100M, and pick the result with "NOR-based" in the summary.

DDJ

Backing Up Isn't Hard To Do

Jerry Pournelle

The Chaos Manor computer system consists of more than a dozen workstations networked into an Active Directory Domain through three Windows 2000 Server systems — one the "master" and the other two online at all times, ready to take over if Imperator fails. It sounds needlessly complex, and really it is: I could do everything we do here with considerably less computing power. But one reason I have a complex system is to help me understand the needs of readers. Many do have small businesses that require a bit more computing power than I typically need. Next month, I'm converting to Windows 2003 Server for the same reason: What I have is more than enough for me, but for many readers it's conversion time and I need to understand their problems.

Most of the actual work here is done on three machines: Lance, which is Roberta's system on which she does e-mail and takes care of sales of her reading software (http://www.readingtlc.com/) and other such matters; Anastasia, my main communications system, which runs Outlook, FrontPage, and does web crawling; and my "Main" machine on which I write all my books, play games, surf the web for web stuff that's likely to be cut-and-pasted into Word documents, and generally do everything else. That machine used to be a D850 RAMBUS system called "Sable"; as we'll see it has been replaced with a new Prescott named "Wendy."

With only three major workstations to worry about, backup ought to be simple: two OUTLOOK.PST files (mine and Roberta's) and all the new Word documents. Provided that OUTLOOK isn't running (you can't copy the OUTLOOK.PST file when OUTLOOK is running; you can extract its information, but that's painful) a couple of batch files will do the job.

Assume that my backup machine in the server room has an F:\ drive, and contains folders for the three machines: Lance, Anastasia, and Wendy. Then all that's required to back up files that aren't open is to open a command window (nee "DOS window"). Assume that Wendy is mapped to the backup machine as W:\; then the command from the backup machine is XCOPY *W:*.doc F:\WENDY\ /e/s/d/y*, which goes out and grabs every .doc file and puts it into the proper folder under F:\WENDY\, creating folders if they don't already exist, and copying only files newer than ones it already sees. Another command seeks out *.pst, again with the /e/s/d/y switches (open a command window and do *xcopy /?* if you don't understand the switches). Run a batch file with both those commands and all Wendy's documents and pst files are backed up. Do the same for Anastasia, and again for Lance, and the job is done. All critical work saved, even assuming I didn't make multiple copies of new creative work at the time it was written.

In practice, it is a little more complex than that. I do have some other critical work, like invoices and accounting records, but that all resides in one folder which is the subject of yet one more *xcopy* command. Still, that's the principle of the thing, and it has been the system I have used for years. All of this could and should live in a single batch file.

The Main Machine Dies

The heading for this ought to refer to the death of a machine named Sable, but I can't quite bring myself to write that; superstition, I suppose. Sable is our two-year old Husky, the dog my kids call "the empty nest pet," meaning she is spoiled rotten. For some unaccountable reason I named the big D850 RAMBUS computer I have used Sable as the main writing machine. I ought to have known better because nothing lasts forever.

I had just installed *Everquest II*, a very large program, from 10 — count them, 10! — CD-ROMs on the D850. The program went out online to get patches and revisions, and up popped a message: It couldn't update some files because the system saw them as read only.

That should have raised some suspicions. I had already had a few flakey moments with that machine; nothing it didn't recover from by turning it off and letting it rest, but there had been reports of voltages out of tolerance (Intel's System Monitor program that comes with Intel motherboards is very good about that), and a couple of times the system didn't boot up properly. I knew it wasn't in 100-percent reliable condition, and I should have taken this read-only thing as a warning to back up everything I possibly could before going any further.

Instead, I used a command line to mark all the files in that folder as R/W. I thought I was being very clever. Wrong. Instead, the machine simply crashed. Attempts to reboot would get part way, sometimes to the Windows XP splash screen, but it never got me to the point that I could copy files. Booting in Safe Mode with or without networking (the system has a ZIP drive and USB ports, so if I could get it running at all, I had ways to sneakernet files out of it) didn't work either. That machine just wasn't going to boot.

The first moral of this story should be obvious. If a system does something entirely unexpected, like telling you that files are read-only when there is no reason for them to be read only, it's a clue. Back that system up, pronto, there's no time to waste.

I didn't do that, and now it wouldn't boot. Well, I thought, what have I lost? It can't be much, because I did copy all my work to other machines, and the last time Larry Niven was over, we copied everything we did to his ZIP disk as well as to Silver, the machine he works on here (Niven likes the Microsoft Natural "humpback" keyboard, so I keep a machine set up with one of those just for him). I did use Sable as the backup machine for accounting, but the primary accounting machine (which runs Windows 98; I wrote that program in CBASIC in 1982 and I see no reason not to use it, but Windows XP doesn't much care for it) was still operational. I quickly went to that one and made backup copies of the accounting files to three other places; no point in tempting fate. But I surely hadn't lost anything else.

Still, it would have been good to be sure, so we put the failed Maxtor DiamondMax disk into another machine as a secondary drive. Alas, Windows sees the disk fine: But it doesn't see any files on it. None whatever, and it wants to format

Jerry is a science-fiction writer and senior contributing editor to BYTE.com. *You can contact him at jerryp@jerrypournelle.com.*

(continued from page 90)
it. We tried several things including Partition Magic, but nothing worked; all the files on that drive seem to have vanished.

So I replaced Sable with Wendy, and began work.

What Was Lost

I soon found what I had lost — a bunch of utilities I use every day. Most, like Spybot Search and Destroy and Adaware were installed on other machines and it didn't take long to bring them over to Wendy. Others, like Notepad Pro, had been on the machine that preceded Anastasia, and on Sable, but I hadn't made other copies, and now were just gone. Of course, I can download that again, and will, but I can't just transfer it, and some of the data files are gone.

Diskmapper is gone. Once again it's not that hard to find it and reinstall it, but I had never backed it up. Some notes I had started in the latest version of Info-Select (nee Tornado Notes) were gone: They use a different format from previous versions. I don't use Info-Select a lot because they keep changing their rules, and now you can't have two copies of the program on the same network and move data between them: That's a deliberate decision on their part, as it was a deliberate decision on mine to stop using the program, although at one time it was a very useful freeform database. One day I'll find something to replace it; indeed, Notepad Pro was the replacement, and while I can restore Notepad Pro, the files of passwords and random data I kept in that format are probably gone.

The Thumbs Plus database of photograph files is missing. This is again no big problem because I keep multiple copies of all photographs, as well as periodically writing them off to a DVD, but it's still annoying. I'll have to reinstall Thumbs Plus and let it spend a couple of hours spidering my photo files.

While I was writing this I found another trivial loss — the Toolbar configuration for Word. Word has gotten complicated enough that getting everything set just right takes a while, and losing that requires you to do it again. The Special Dictionary, built up over the years, was gone. I have a separate .dic file for every novel (after all, I make up words and have alien names in my stories, none of which I need in my nonfiction dictionary), and there's a pretty good special dictionary over on Anastasia; for that matter I can consolidate dictionaries from years ago. Still, it takes doing, and that too is something I should have backed up and didn't.

I could continue the list but surely the point is made? In addition to all the things I thought of, there were many other files I ought to have been backing up and didn't.

The New Backup System

Clearly, what I should have been doing was backing up files from a central point using the network. There's a problem with that: You can't copy open files, and there's never a time when I'm certain to be away from my desk. When I can't sleep, rather than practice lying in bed staying awake, I just get up and do some work or read a book, or even play FreeCell.

Anastasia already has a backup system. The primary drive is a RAID 1, which is a pair of disks in a mirrored array. Both would have to fail before I lose all the data. The next step is to do that for my main writing machine, and I will. Even so, this only protects me from hard drive failures. It does nothing for operator error.

On the other hand, we've had a great backup system for a single machine going on for weeks: The CMS external USB 2.0 backup system that was installed on Silver, the machine that Niven and other visitors use. It was put there because it was a convenient place for it; but it has worked so well I am tempted to take it from Silver, reformat the disk, and install it on Wendy. When I mentioned that to CMS they made me a better offer: They're sending a couple more units, and I'll put one on Roberta's machine, and one on Wendy, and that way both our main machines will be backed up automatically and without our having to worry about it.

The CMS backup software nags you if you forget to make a backup; but it does it politely, and it works in the background so that on fast machines — in our case, Prescotts — you don't notice that it's working at all. The more I use this, the more enthusiastic I have become, and I only wish now I had installed the first CMS unit on Sable before her drive went west.

Live and learn. No one gets the backup religion until there's a disaster. In our case it was a lot short of disaster, but I'm not waiting for a more serious warning. We're doing a lot of backup now, but I won't be happy until it's automatic.

I still wish that Seagate, which bought Palindrome, would get that wonderful backup management program out again. It really worked, doing everything you ever wanted a backup management system to do.

Winding Down

The game of the month is *Everquest II*, which I've had fun with. The book of the month is David McCullough's *John Adams*, a very readable account of one of the most intellectual of the Founding Fathers. Without Washington's charisma and leadership we would never have been able to form the Union, but without Adams and his legal sense it is not likely to have held together long. Washington, Hamilton, and Adams were all pretty essential in the making of the New Order of ordered liberty. Incidentally, although Adams supported the Sedition Act, which allowed punishment and suppression of "seditious libel." He would have been the first to denounce the new Treasury regulations requiring publishers in the United States to obtain a federal license before publishing dissenting works condemned by their home countries — regulations that would have required, for instance, a license to publish *Dr. Zhivago*, or dissenting works out of Cuba. Punishing sedition after publication upon proof that it is seditious libel is nowhere near the same thing as requiring a license before publication: Whatever else the First Amendment was intended to protect, it most certainly abolished the very notion of prior restraint, and Adams would have been among the most vigorous opponents of anything like federal licensing for publications.

Another book of the month is Neal Stephenson's three-volume series ending with *The System of the World*. This giant exposition into the 17th and early 18th Centuries is a tour de force. Parts of it are hard to read and some of it is needless, but it's certainly a worthwhile experience.

The first computer book of the month is James C. Foster and Steven C. Foster, *Programmer's Ultimate Security DeskRef* (Syngress, 2004; ISBN 1932266720). Encyclopedic in form, it is precisely what the title says it is.

The second computer book of the month is a pair of O'Reilly books in their "Hacks" series: Sid Steward's *PDF Hacks* (O'Reilly & Associate, 2004; ISBN 0596006551), and Shannon Sofield's *Paypal Hacks* (O'Reilly & Associate, 2004; ISBN 0596007515). I've mentioned before that Amazon and Lightning Press have a system through which you can sell documents through the Amazon store. Basically, you buy ISBNs from Bowker, put your documents in PDF, Microsoft Reader, and other formats as you choose, set a price, and upload to Lightning, after which you're a publisher, and your works are listed in the Amazon index. Francis Hamit has been doing this with his magazine articles and other publications, and while sales build slowly, they do build. The *PDF Hacks* book is useful in getting material into the right format; anyone who works with PDF needs this book. The Paypal books told me a lot about using Paypal, and I'll be incorporating some of that into my web site.

DDJ

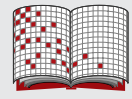
Books for Developers

Douglas Reilly

Many books are available that cover specific technologies, such as ASP.NET and C#. Many fewer that address the development process are available. Fortunately, two recently released books address just this topic. Mike Gunderloy's *Coder to Developer* covers what is required to move a newly minted programmer from being merely a coder into a more complete developer. A complete developer is someone who knows not only the syntax of a programming language or two, but also can take that knowledge and create full-featured and reliable applications. Anyone who has been coding for a while recognizes that even if we thought that we were complete developers right out of school or after our first programming success, we had a lot to learn. Moreover, code we developed even five years ago often makes us cringe as we look at it with the benefit of the additional five years of learning. *Coder to Developer* helps speed you along in that journey.

Coder to Developer steps you through the process of developing software, starting with the often overlooked planning phase. Gunderloy also guides you through the use of source-code control and many other tools that are, at best, mentioned in passing during most formal training. It is the proper use of the appropriate tools that often separates a coder from a more complete developer.

Another topic that often is given less than complete coverage in most training settings is the need to track and handle bugs in our code. An especially important chapter of the book discusses the need and benefit of logging application activity. This is something I learned through lots of difficult experience with medical applications that ran 24/7, at hours I could not be present to observe. If all beginning developers take away from this book is the need to log significant system activities, it is worth the price of admission.



Coder to Developer: Tools and Strategies for Delivering Your Software
Mike Gunderloy
Sybex, 2004
352 pp., $29.99
ISBN 078214327X

Code Complete, Second Edition
Steve McConnell
Microsoft Press, 2004
960 pp., $49.99
ISBN 0735619670

There are a couple of things you should know before you commit to *Coder to Developer*. First, this is a book that very much focuses on Visual Studio .NET. While much of the advice is appropriate for all developers, many of the specific examples and suggestions are for .NET programmers. In addition, some of the specific recommendations may change for Visual Studio 2005. Keeping in mind that we are likely more than a year away from general availability of a released version of Visual Studio 2005, *Coder to Developer* will continue to be a worthwhile purchase for .NET developers.

Code Complete, Second Edition, is the most recent edition of Steve McConnell's classic *Code Complete,* originally released in 1993. If you're familiar with the original, this edition is not a complete rewrite. Much of the first edition's content holds up very well, even all these years later. McConnell has, however, improved upon the first version in many ways. First, many of the "Hard Data" notes point to more recent studies and articles. The "Hard Data" notes were among portions of the original book that I quoted when dealing with management; folks who did not understand that increasing capital expenditures, providing better working conditions, and so on, would be worth the expense. I did not always succeed, but I had a better argument than just, "I think better conditions would help the staff program better."

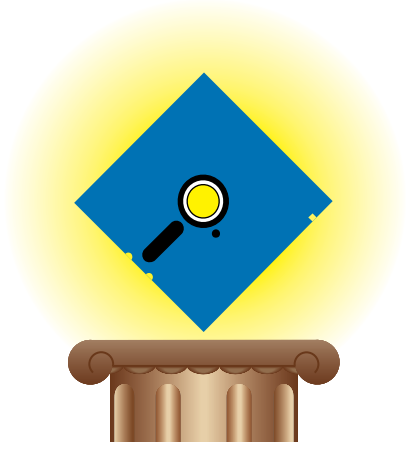
I did not do a careful side-by-side comparison with the first edition. That said, there are two more chapters in the new edition, and some more recent topics, such as agile development, team programming, and refactoring are covered well. The chapters on naming classes, variables, constants, and so on, could have been combined to eliminate some duplication, but I can live with the amount of duplication present.

It is a little ironic that *Code Complete* from Microsoft Press would in fact be much more platform agnostic than *Coder to Developer*, published by Sybex. *Code Complete* is a book that you should have on your bookshelf. Examples are as often in Java as C++ or Visual Basic, and in most cases the specific language does not even matter. Since this was a new edition, I expected to see more examples using C#, and in that respect I was disappointed. In a few cases where the topic cried out for a C# example (for instance, the *foreach* loop), I did find the example in C#. Most of the Java or C++ examples will be useful for C# developers, but an understanding of C++ or Java will not hurt, since if an example uses the C++ namespace resolution operator, you will need to understand the C# equivalent.

Which of these books for developers should be on your bookshelf? That depends on your current situation. If you are a developer moving to .NET development and need specific guidance on tools and techniques, *Coder to Developer* won't disappoint you. The specific tools coverage Gunderloy provides is timely and unique. If you are a Java developer, or a developer in a number of languages and platforms, *Code Complete* will serve you well. The variety of example languages, as well as the numerous references, is a welcome addition to any developer's bookshelf.

DDJ

Douglas is president of Access Microsystems and can be contacted at doug@accessmicrosystems.com.



Simian Systems has developed phpBeans, a standard for implementing enterprise-level applications in PHP. phpBeans defines a method of allowing objects to communicate seamlessly across separate machines. phpBeans provides specifications and reference implementations of the phpBeans Object Server, the phpBeans Protocol, and the phpBeans Client API.

Simian Systems
1071 Corydon Avenue
Winnipeg, Manitoba
Canada R3M 0X3
204-942-8630
http://www.simian.ca/

Red Gate Software has released a new version of its SQL Bundle Developer Edition, designed to simplify development of automated programs for comparing, synchronizing, and packaging Microsoft SQL Server databases. The new release includes compression for the SQL Packager that reduces package sizes. SQL Bundle Developer Edition comprises five Red Gate packages: SQL Compare, SQL Data Compare, DTS Compare, SQL Packager, and SQL Comparison and Synchronization Toolkit.

Red Gate Software
St John's Innovation Centre
Cowley Road, Cambridge
United Kingdom CB4 0WS
+44 870 160 0037
http://www.red-gate.com/

Wind River Systems has announced a number of new releases for device development and support. Wind River Workbench 2.2 is an Eclipse-based development suite supporting VxWorks, Linux, and in-house operating systems. The Wind River General Purpose Platform integrates VxWorks 6.0, Wind River Workbench 2.2, and middleware. The Wind River Platform for Network Equipment, Linux Edition combines Carrier Grade Linux with support for telecom hardware. Lastly, Wind River Market Specific Platforms include industry specific middleware for consumer devices, industrial devices, and network equipment.

Wind River
201 Moffett Park Drive
Sunnyvale, CA 94089
408-542-1322
http://www.windriver.com/

Caphyon LLC has released Advanced Installer Professional 2.1, a Windows Installer authoring tool for MSI packages. Advanced Installer runs on Windows 2000/XP, and the packages it creates run on all Microsoft Windows 9x/ME/NT/2k/XP operating systems. The project files are saved as XML. This new version adds support for installation folders that are synchronized with disk folders for easier project maintenance.

Caphyon LLC
2017 California Street, Suite 1B
Mountain View, CA 94040
http://www.caphyon.com/

Codejock Software is shipping its Xtreme Toolkit Professional Edition 9.51 for Visual Studio .NET, including Xtreme Suite, Xtreme Command Bars, Xtreme Docking Pane, and Xtreme Property Grid. The Xtreme component family provides customizable UI components for developing Microsoft-style applications for use with Microsoft Foundation Class (MFC), ActiveX, and Microsoft.NET development platforms.

Codejock Software
204 W Exchange Street, 2nd Floor
P.O. Box 726
Owosso, MI 48867
989-723-1442
http://www.codejock.com/

ActiveState's Perl Dev Kit 6.0 introduces GUIs for most tools, providing visual guides to build options. Also included is the PDK Filter Builder, a tool for interactively creating text processors that do custom filtering and replacement operations, and VBScript Converter for generating Perl code from VBScript. The Dynamic DLL Loader supposedly eliminates the need to write temporary files to the disk during execution of most applications, saving disk space, reducing clean-up of temporary files, and increasing security.

ActiveState
580 Granville Street
Vancouver, BC Canada V6C 1W6
604-484-6800
http://www.activestate.com/

Canoo has released a new Eclipse 3.0 plug-in that simplifies Internet-application development with the UltraLightClient Java library. Offering a server-side programming and execution model, UltraLightClient complements the Eclipse Rich Client Platform (RCP) to help you provide responsive GUIs for web applications within J2EE and J2SE infrastructures. UltraLightClient follows the Swing API but takes care of the code split and optimizes communication.

Canoo Engineering AG
Kirschgartenstrasse 7
4051 Basel, Switzerland
+41 (61) 228 94 44
http://www.canoo.com/

Wingware has upgraded the Wing IDE for Python to Version 2.0. New features include a completely redesigned customizable user interface, call tips, syntax error indicators, editor tabs and splits, multifile wildcard and regular expression searching, integrated documentation and tutorial, a German localization, and Unicode support. Wing IDE Professional is available on Windows, Linux, Mac OS X, and other platforms.

Wingware
P.O. Box 1937
Brookline, MA 02446-0016
617-232-0059
http://wingware.com/

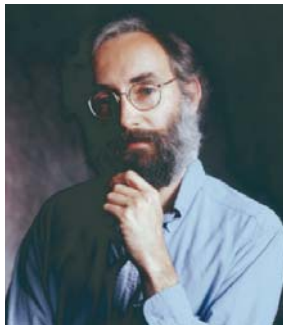
IDEAL Software is shipping Version 3.60 of its ad hoc document creation library Virtual Print Engine (VPE) for Windows. The new version ships with two new .NET components, one for Winforms and one especially created for ASP.NET. The components let you dynamically create precise reports, documents, forms, and drawings by calling functions at runtime.

IDEAL Software GmbH
Erftstrasse 102 a
41460 Neuss, Germany
+49 2131 1511 - 690
http://www.IdealSoftware.com/

Perforce Software has added a Time-lapse View to its Perforce Software Configuration Management system. Time-lapse View provides a graphical way to view the complete change history of individual text files stored in the Perforce repository. Perforce also now integrates with Discreet 3ds max, Adobe Photoshop, and Alias Maya.

Perforce Software Inc.
2320 Blanding Avenue
Alameda, CA 94501
510-864-7400
http://www.perforce.com/

DDJ



Last Year Went Like Clockwork

This one is for the nattering nabobs of nadsat.

2004 was a horrorshow year for Steve Jobs, the CEO of Apple and Pixar. One time, all the gazettas called him a nadmenny baddiwad malchick, but in 2004, it was a different raskazz. It's all about the pretty polly, my droogies.

Our Steve shvatted more of the stuff than any other CEO in Silly Con Valley in Anno Domino 2004. He was the bolshy bugatty in Forbes gazetta every month. Apple's stock rose like a koshka up a zvonock. Every lewdie's glazz was on Steven Pee Jobs.

Steve was the CEO with the two rabbits, and each of these rabbits was always shvatting lots of pretty polly by prodding techie vellocet for the lewdies to mounch. From the Apple rabbit came carman-sized veshches for razrezzing warbles and dorogoy veshches with knopkas to fist, and from the Pixar rabbit came bolshy animated sinnies.

And didn't all the malenky nadsats tolchock pee and em for the gollies to viddy Pixar's latest raskazz at the sinny? And didn't they fill Steve's carman with the pretty polly? They did and they did.

But oh my droogies, sloosh this: The messel that warms Steve's heart and that is like the great Ludwig Van to his ookos is not the pretty polly in his carman, but the fact that the lewdies of Woodside are finally letting him crack his domy. Steve and his family were all oddy knocky in the fourteen bedrooms and thirteen and a half baths of that starry staja and bezoomny to ookadeet the place.

2004 was also the year that Dan Gillmor may have traded a creech for a chumble by ookadeeting his rabbit at the starry San Jose gazetta to become a Citizen Gazettista. Pursuing his sneety, our Dan was, even if he could snuff it. He might be gloopy, the sarky lewdies at the gazetta platched, but the malchick's got gulliwuts.

But sloosh me horrorshow, my droogies: Dan may not be out of his gulliver after all. The time may be dobby for fillying with blogs and dratsing with the gazettas. Because those malchicks at the bolshy gazettas seem to be only a dook of their former selves, either gloopy or spoogy, without a dobby messel, prodding their baddiwad vellocet for the lewdies and getting all grahzny with the buggaties.

But that's just my nadmenny opinion.

IBM also pulled an ookadeet in 2004, letting the Chinese kupet its entire pee sea rabbit. This snuffed a raskazz that began in 1981, when the starry computer company got too droogy with Bill Gates and let him crast its dorogoys. Of course, that's just how IBM viddies it. Doubtless Bill viddies it differently. You have a choice of nadmennies, as always.

In short, my droogies, it was a bezoomny year. This one looks like another.

Anthony Burgess helpfully provided a dictionary for nadsat, the invented patois in which he wrote his novel *A Clockwork Orange*. Many copies of that dictionary can be found on the Internet; here are my versions of the definitions of the nadsat terms used in this month's column:

baddiwad: *bad*
bezoomny: mad
bolshy: *big, great*
bugatty: *rich, rich person*
carman: *pocket*
crack: *to break up, tear down*
crast: *to rob, to steal*
creech: *to shout, to scream*
chumble: *to mumble*
dobby: *good, right*
domy: *house*
dook: *trace, ghost*
dorogoy: *valuable*
dratse: *fight*
droog: *friend*
filly: *to play or fool with*
fist: *to punch*
gazetta: *newspaper, magazine*
glazz: *eye*
gloopy: *stupid*
golly: *unit of money*
grahzny: *dirty*
gulliver: *head*
gulliwuts: *guts*
horrorshow: *good, well*
knopka: *button*
koshka: *cat*
kupet: *buy*
lewdies: *people*
malchick: *boy*
malenky: *little, tiny*
messel: *thought, fancy*
mounch: *snack, to snack*
nadmenny: *arrogant*
nadsat: *teenage, teenager*
oddy knocky: *lonesome*
ookadeet: *to leave*
ooko: *ear*
pee and em: *parents*
platch: *to cry*
pretty polly: *money*
prod: *to produce*
rabbit: *work, job*
raskazz: *story*
razrez: *to rip*
sarky: *sarcastic*
shvat: *to grab*
sinny: *cinema*
sloosh: *to hear, to listen*
sneety: *dream*
snuff: *to kill*
snuff it: *to die*
spoogy: *terrified*
staja: *state jail, prison*
starry: *ancient*
tolchock: *to hit or push*
vellocet: *drug*
veshch: *thing*
viddy: *to see, to look*
vred: *to harm, to damage*
warble: *song*
zvonock: *bellpull*

Michael Swaine
editor-at-large
mike@swaine.com