
We Trip The Light
FANTASTIC

CMP
United Business Media

C/C++ • .NET • Java • Linux • C# • Win32

#373 JUNE 2005

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

http://www.ddj.com

TESTING & DEBUGGING

Omniscient Debugging

Examining Software Testing Tools

Dissecting Error Messages

Debugging Production Software

Performance Monitoring

System Verification with SCV

Portability & Data Management

Inside TR1 C++ Library Extensions

Software Reuse: Does It Really Pay Off?

Loadable Modules & The Linux 2.6 Kernel

ASP.NET & Multiplatform Environments

Swaine on VB-not-net

XScale, C++, & Hardware-Assisted Breakpoints

Ed Nisley on Security

Jerry Pournelle on Spyware

CONTENTS

JUNE 2005 VOLUME 30, ISSUE 6

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 light up July with our coverage of Java.

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/

Quirky Is Fine With Me

Quirky is fine with me. I'm a fan of all things slightly out of whack—people, places, movies, museums, books, you name it. My buddy Michael once took to wearing flea collars around his neck, wrists, and ankles when his apartment was invaded by fleas. Among my favorite books are *The Tulip: The Story of a Flower That Has Made Men Mad* and *The Dog in British Poetry*. Then there's the Bily Brothers Clock Museum in Spillville, Iowa, and the movie *Dancing Outlaw*.

The latest to join the quirky club is *The Collector's Guide to Vintage Intel Microchips*, by George M. Phillips, Jr.—an e-book (in PDF) on CD-ROM that has everything you ever wanted to know (and then some) about Intel processors, controllers, RAM, ROM, EPROMs, memory, support circuits, and the like. This includes 1300 pages worth of part numbers, photographs, data sheets, the names of the designers (and interviews with them, in some cases), and occasionally, the collectible value of the chip—all indexed, cross-linked, and in color (see http://www.vintagemicrochips.com/).

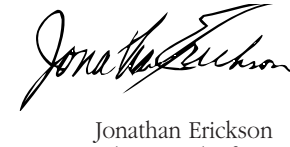
Actually, George isn't alone in his fascination with silicon and circuits. It turns out that there's a worldwide network that collects CPUs and microchips. For instance, in Poland Marcin Majewski hosts his ABC CPU (http://www.abc-cpu.xn.pl/); in Germany, Christian Lederer puts up the CPU Museum (http://www.cpu-museum.de/); in Oregon, John Culver is the curator of the CPU Shack (http://www.cpushack.net/); in Corsica, Desideriu maintains the CPU Museu (http://cpu-museu.net/); and Lee Gallanger hosts his Vintage Chip Trader (http://www.vintagechiptrader.com/).

But when it comes down to it, collecting vintage microchips is no quirkier than collecting, say, vintage vacuum tubes. Bob Deuel's collection, for instance, consists of tens of thousands of vacuum tubes, although he displays *only* 1200 or so in his home, including a 228-pound tube from a 50,000-watt broadcast radio transmitter. And yes, Bob's not alone out there, at least when it comes to vacuum tubes. There's the Tubepedia (http://www.aade.com/tubepedia/1collection/tubepedia.htm); the Tube Collectors Association (http://www.tubecollectors.org/); Kilokat's antique light bulb site (http://www.bulbcollector.com/); Mike's Electric Stuff (http://www.electricstuff.co.uk/); Ake's Tubedata (http://www.tubedata.com/); and more.

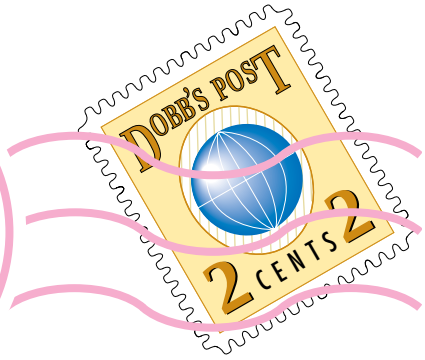
But as you can see in *The Collector's Guide to Vintage Intel Microchips*, there's more to serious collecting than grabbing a microcontroller here and a vacuum tube there and dropping them into old cigar boxes (which, by the way, are also collectible; see http://galleries.cigarweekly.com/RickMG/album01). There's all that information to be gathered—specifications, packaging, part numbers, and more. Some of the most difficult information for George to compile was the introduction dates of the more than 300 different chips Intel introduced *before* 1980. Recall that the 8008 was introduced in 1972, and Intel's first microchip, the 3101 static RAM, was introduced around 1969. Of course, the easy way out would have been to just ask Intel, which maintains its own chip museum (http://www.intel.com/museum/). Unfortunately, it didn't occur to Intel to begin documenting exact introduction dates of its chips until the mid 1980s. By then, many of the introduction dates could only be determined by asking engineers who had worked for Intel in the 1970s if they remembered when particular chips were introduced. The answer George usually got was something like, "I think we worked on that in late 1971, but it could have been 1972, or was it 1973?" He adds that it is even more difficult when tracking down introduction dates of the different versions of a chip—the 2107, 2107A, 2107B, 2107C, and so on.

Alternatively, you'd think you could just go to the library and simply peruse old Intel data catalogs. Alas, libraries don't have them and early Intel data catalogs are almost as rare as Gutenberg Bibles. There's only one known copy of Intel's first data catalog printed in September 1972, for instance, and no known copies of the 1973 or 1974 data catalogs. This means (you guessed it) early Intel data catalogs are collectible, too, and it's not uncommon for them to sell for hundreds of dollars on eBay. Early Intel MCS-4 and MCS-8 user manuals, memory books, sales brochures, and the like are also sought after by collectors. It's no surprise that it took George about five years to put the e-book together.

So what possesses someone to devote this much time and energy into putting together such a quirky project? "Maybe the best answer is because it matters," says George. He goes on to explain that he's been a programmer for 20 years and seen how computers have changed the world. However, he says that people who live through events often don't realize the significance of those events as they are happening. It's all still too new. But George believes that future generations will look back on this time as a monumental turning point in history, on par with the discovery of fire and invention of writing—if they have a historical source to turn to. After all, no one ever said that just because something is quirky that it isn't important.



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



OAF Again

Dear *DDJ*,

I have been looking at John Trono's OAF algorithm ("Applying the Overtake & Feedback Algorithm," *DDJ*, February 2004) with great interest. This is a fantastic piece of work, which I am very keen to understand and hopefully apply, and was wondering if John could please explain the following sentence with regards to working out the "Distance" factor a bit further: "…where distance is the largest, integer multiple of SQRTN less than or equal to the separation between the two team's rankings, which in this case is floor(57/11) or 5." For example, if there are 20 teams in my league and the two teams involved are ranked 3 and 12, what would the distance be? I live in Manchester, England, and was wondering whether I would be able to use this method to create some soccer ratings? The winning score margins for each game are going to be a lot lower than for American football, the most frequent being just 1 or 2. Should I multiply these scores by another constant to fit them into the algorithm? Also what do you think should happen to each teams rating if a Draw (Tie) result occurs?

Ian Broughton
tracy.jones63@ntlworld.com

John responds: Ian, thank you for your interest in (and kind words about) the OAF algorithm. Though I left out how ties games are handled in the brief summary that is on my web page (http://academics.smcvt.edu/jtrono/OAF.html) (since they no longer are a possibility in NCAA football), such an outcome is treated almost like a loss. When two teams' ratings are averaged in OAF, the victor gets the average +0.4 and the loser the average –0.4 as their updated ratings. When a tie occurs, the team that had the higher rating before the tie game has 0.4 added to the average, and the other team receives the average –0.4 as its new rating.

If there are 20 teams, then the integer used for $\sqrt{N}$ is the floor of the rounded result of $\sqrt{(20)}$, which in this case is 4. If a team ranked as #3 loses to a team that is ranked #12, then the modified update rule would compute that team #12 is (12–3)/4=2 "intervals" outside of the region where the pure OAF update would be applied and so the denominator of 4 would be used instead of 2, when computing the "average" of the two opponents' ratings. (Teams 4–6 would use a denominator of 2, i.e., the true mean, 7–10 would use a denominator of 3, 11–14 would use 4, etc.) I hope this clears that up for you. Please go back and reread the last three paragraphs on that web page to see if you follow what I describe here. And remember, there is the caveat about using the larger update of the two that are computed, as mentioned in the penultimate paragraph on my web page that you quote from below.

I would be curious to hear about your results when applying OAF to the soccer games in England. I don't think you would have to do anything special to adjust the scores because of the smaller differentials; you might just end up with ratings that are closer, that's all. (Since touchdowns in football are like goals in soccer, you might try multiplying all your scores by 7, but again, I am not sure if that would be necessary.)

Good luck; don't hesitate to ask for clarification if my description above is unclear, and keep me posted about how your study goes when applying this to soccer.

Silent Update

Dear *DDJ*,

I appreciate the skill that went into Zuoliu Ding's article "A Silent Component Update for Internet Explorer (*DDJ*, April 2005). As a user, I ask, how do I prevent a software company or malicious agent from silently updating the software on my hard drive? Will a firewall like ZoneAlarm do?

Bill Weitze
bweitze@california.com

Zuoliu responds: Bill, thanks for your interest in my article. Regarding your question, I would like to say there is nearly no way to prevent such an update with a third-party product. This is because such a silent update is triggered by the specific software that has been installed in your local machine based on your trust already. The download and update is guided by the software resided. If a hacker hijacked such a company download site and knew the update mechanism, your computer would be vulnerable and defenseless. For some general invasion, you may take [precautions] like blocking some ActiveX downloads from the prompts by XP/SP2. But for such a silent update, the individual company should take full responsibility in designing secure updates, such as encoding download information, dynamizing update sites, and so on. Hope this helps.

Is Wind Power Hot Air?

Dear *DDJ*,

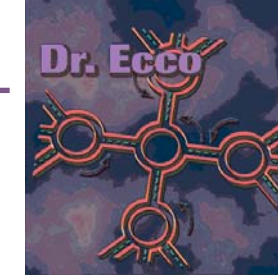
As a part of Jonathan Erickson's "looking back" perspective in the February 2005 issue of *DDJ*, he effused about the amount of energy being created via wind-generated electricity. However, nuclear energy is an idea whose time has come and gone—and is come again with a vengeance. Simply put, it is the cleanest, safest, best way to create energy for people on this Earth. Wind power doesn't even come close to fitting the needs of this world. Note that a "wind farm" would require about 300 square miles of space to produce the equivalent output of one nuclear plant. The wind turbines slice-and-dice birds with a terrible regularity.

Don't take my word for it. China researched it and is now committed to building a minimum of 25 new reactors in the next decade. All completely fail-safe. The design is called "Pebble Bed Reactors." You can fire these puppies up and walk away from them forever. They won't blow up and they won't melt down. I refer you to the February 2005 issue of *Wired* magazine, which covered this issue in some detail. The biggest threat to the reintroduction of nuclear power to America is, of course, the Luddites—people who believe that "nuclear is bad." Again, *Wired* covers these types pretty well. And so I ask Jonathan to reconsider the scientific evidence that bodes extremely well for nuclear power, and drop the hippie-cult thing with "wind power." In the end, "Wind Power" is just a bunch of hot air.

Peter Andre
peter.seattle@gmail.com

Jonathan responds: Thanks for your note, Peter. Did I say that windpower ought to replace nuclear power? I don't recall doing so. Should there be alternatives to coal-based power plants? Yes. Coal-based plants screw up the environment when you dig the coal and when you burn it. I don't have a problem with nuclear power. But I also think that places rich in wind (Washington DC doesn't count) can harvest that energy and provide an economic boon to the local economy. This includes, say, western Kansas, the Dakotas, and other such windblown areas. It probably isn't feasible to build nuclear plants in western Kansas, but wind turbines do make sense.

DDJ



The Pirates' Cantilever

Dennis E. Shasha

The tall man who came knocking at Ecco's apartment cut quite a figure. A titanium prosthetic had replaced his left leg below the knee, but this big man made his way around with a vigor that few fully legged men could match. He wore a dark bushy beard and shaggy hair. He looked quite scary except that his dark eyes twinkled and he sported very distinctive smile lines.

"Call me Scooper," he said by way of introduction. "After my, er, accident, I got this leg and started walking around with a wooden cane. Soon after, my buddies gave me an eye patch, joking that I should put a jolly roger on my jeep's antenna. I had some time on my hands, so I decided I would study pirates. I found some pretty modern ones, operating as late as the 1930s off the Carolina coasts and specializing in attacking yachts. They did this with stealth and intelligence. Never did they physically harm any one. Sometimes their booty was difficult to divide, however.

"In one heist for which I have partial records, almost half the value of the theft was embodied in a single, very well-known diamond — so well known in fact that they couldn't sell it right away. They decided to award the diamond to the pirate who could win the following contest. They were very mathematically adept pirates, so I'm convinced someone won. I want to know how.

"Here is the contest: There is a set of wooden planks *P*. Given *P*, each contestant is to construct a structure of planks that cantilever off a flat dock.

"Is someone going to walk the plank?" 11-year-old Tyler asked.

"No," Scooper replied with a smile. "They didn't want anyone to walk the plank yet, but they wanted to make the pile extend out as far as possible from the dock without requiring glue, ropes, or attachment. That is, they wanted to create a pile of planks and have them stay put assuming no vibrations or wind."

"If all the planks are the same size and weight, and each plank can sit on only one other plank and support only one plank, then this is the 'Book Stacking Problem' and is nicely analyzed on http://mathworld.wolfram.com/BookStackingProblem.html," Liane volunteered.

"Interesting," Scooper said. "Could you explain how that goes?"

"The basic idea is simple," Liane replied. "One plank can extend half-way out in a cantilever without tipping; if you have two planks, then the first one extends 1/4 the way out and the other extends 1/2 way out beyond the first as you can see in the drawing in Figure 1.

"Let's analyze this. Suppose each plank weighs *W* and is of length *L*. The net weight of each plank is at its center. The torque around the dock edge is therefore $+(L/4)W$ due to the bottom plank and $-(L/4)W$ because of the top plank. So the net torque is zero. Similarly, the center of the top plank rests on the outer edge of the bottom plank, so it will not flip over. More planks allow an arbitrarily long cantilever, given enough planks."

"Physics is always a surprise," Scooper said.

"1. Our pirates are not so bookish, however. They have 10 thick and rigid planks of lengths 1 through 10 meters and weighing 5 through 50 kilograms they had just captured from a French yacht. Further, according to their description of the contest, all planks should share the same orientation — they should lie horizontally and their lengths should be perpendicular to the dock edge. To start with, they required that a plank could lie on only one other plank (or on the dock) and could support only one other plank. In that case, how far from the dock can a plank extend without tipping for the best structure?

"2. Now suppose that the pirates allow two or even more planks to lie on a supporting plank. Can you create a cantilever that goes even farther out?" (*Hint:* Yes, by more than a couple of meters.)

Liane and Tyler worked on these for an hour before coming up with an answer.

"Nice work," Scooper said, looking at their drawings. "Here is the hardest problem. Suppose there are no constraints about how planks can lie on one another? They need not share the same orientation, many planks can support a single plank, and so on. Then how far can the cantilever go out using just these 10 planks?"

I never heard the answer to this one.

For the solution to last month's puzzle, see page 92.

DDJ

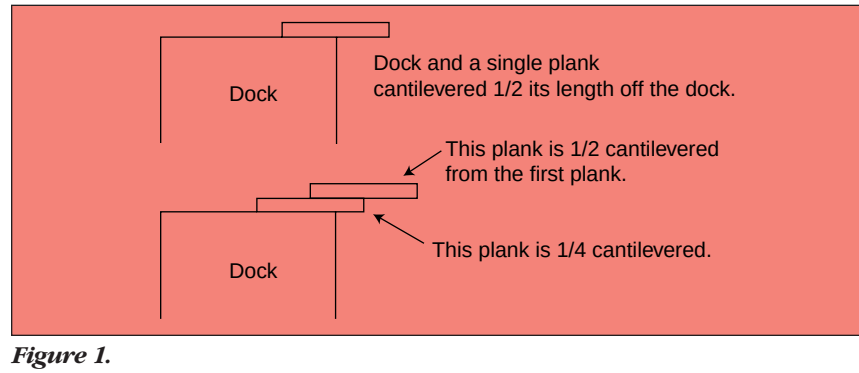


Figure 1.

Dennis is a professor of computer science at New York University. His most recent books are Dr. Ecco's Cyberpuzzles *(2002) and* Puzzling Adventures *(2005), both published by W. W. Norton. He can be contacted at DrEcco@ddj.com.*

Natural Language to Source Code

Researchers at the Massachusetts Institute of Technology are developing a language-to-code visualizer tool called "Metafor" that translates natural languages (like English) to source code. According to researchers Hugo Liu and Henry Lieberman, Metafor is a "brainstorming" editor that "interactively converts English sentences to partially specified program code, to be used as 'scaffolding' for a more detailed program"— an outliner, in other words. Metafor builds program skeletons in Python and other languages from parts of speech and language syntax, in which noun phrases are equivalent to objects, verbs to functions, and adjectives to object properties. A parser identifies the subjects, verbs, and object roles, and the Metafor software maps these English language constructs to code structures. For more information, see http://web.media.mit.edu/~hugo/publications/papers/CHI2005-NLInterfaces.pdf.

Blue Gene Blazes On

IBM's supercomputer-in-progress, Blue Gene/L, has eclipsed its own performance record. The partially assembled system was ranked the fastest in the world last November, when it performed 70.72 teraflops on the Linpack benchmark. Now, although the cluster is still only half finished, it has nearly doubled in speed— clocking in at 135.3 teraflops on the same benchmark, according to the Department of Energy. IBM estimates that when Blue Gene/L is complete, it will be capable of an unprecedented 360 teraflops (http://www.research.ibm.com/bluegene/). When finished, Blue Gene/L will consist of 65,536 dual Power PC 400 cores running at 700 MHz—131,072 processors—with on-chip memory, and two dual floating-point units to speed calculation. The system will be densely packaged into 64 racks and integrated with multiple interconnection networks. The Blue Gene/L supercomputer is being installed at Lawrence Livermore National Labs, where it will perform nuclear weapons simulations.

Hitachi Shows New Robot

Hitachi's EMIEW—"Excellent Mobility and Interactive Existence as Workmate"— design is a walking, talking humanoid robot designed to give Honda's ASIMO a run for its money. Hitachi has built two EMIEWs, dubbed "Pal" and "Chum." The wheeled EMIEWs can move twice as fast as ASIMO, although unlike the other robot, they can't climb stairs. While EMIEW may seem to be playing catch-up to ASIMO and Sony's QRIO, the company is quick to point out that it has dabbled in nonindustrial robots before. Its first bipedal robot was shown at the 1985 Tsukuba Expo in Japan. For the EMIEWs, however, Hitachi chose to use a two-wheeled base resembling a Segway, which lets the robots keep up with a normal human's walking pace.

The EMIEWs also feature two arms and hands capable of grasping objects, as well as humanoid torsos and heads. They can talk, and are capable of engaging in dialogue with humans, although their vocabularies are limited to about 100 words. Hitachi estimates that in five to six years of linguistic training, Pal and Chum could be ready for practical jobs as information desk workers or office support staff.

Ultra-Fast Electrical Signals Captured

Researchers at the University of California, Los Angeles have for the first time captured and digitized electrical signals at the rate of 1 trillion times per second. Professor Bahram Jalali and graduate researcher Yan Han have developed a one-tera-sample-per-second single-shot digitizer that lets scientists see, analyze, and understand lightning-quick pulses. The one-tera-sample-per-second single-shot digitizer uses light to first slow down the electrical waveforms, allowing the ultra-fast waveforms to be digitized at pico-second intervals— or one-millionth of one-millionth of a second. One application being studied is the development of defenses against microwave "e-bombs" that can destroy electronic devices. For more information, see http://www.newsroom.ucla.edu/page.asp?RelNum=6000.

Intel Science Talent Search Winners Announced

David Vigliarolo Bauer of Bronx, New York, has been awarded a $100,000 scholarship for being named the first-place winner of the 2005 Intel Science Talent Search (Intel STS) competition. Bauer, of Hunter College High School, designed a new method using "quantum dots" (florescent nanocrystals) to detect toxic agents that affect the nervous system. Second place and a $75,000 scholarship went to Timothy Frank Credo of the Illinois Mathematics and Science Academy in Highland Park, Illinois, for developing a more precise method to measure very brief intervals of time— picoseconds (trillionths of seconds)— over which charged secondary particles of light travel. The $50,000 third-place scholarship went to Kelley Harris of C.K. McClatchy High School in Sacramento, California, for her work on Z-DNA binding proteins, which may play a role in cell responses to certain virus infections. All in all, more than 1600 entries were submitted by students ranging in age from 15 to 18, with Intel awarding a total of $580,000 in prizes. For more information, see http://www.intel.com/education.

Eclipse Roadmap Released

The Eclipse Foundation has released Version 1.0 of its Eclipse Roadmap (http://www.eclipse.org/org/councils/roadmap.html), a document that outlines future directions of Eclipse. Among the themes and priorities outlined are issues related to scalability and enterprise-readiness; simplicity and extensibility; globalization; and attention to the rich client platform. Among the projects the organization will likely launch over the next year are: more extensive coverage of the software development lifecycle, embedded development, multiple language support, and vertical market technology frameworks.

New Largest Known Prime Number Discovered

Martin Nowak, an eye surgeon in Germany, and a long-time volunteer in the Great Internet Mersenne Prime Search (GIMPS) distributed computing project (http://www.mersenne.org/prime.htm), has discovered the largest known prime number. Nowak used one of his business PCs and free software by George Woltman and Scott Kurowski. His computer is a part of a worldwide array of tens of thousands of computers working together to make this discovery. The formula for the new prime number is $2^{25,964,951}-1$. The number belongs to a special class of rare prime numbers called "Mersenne primes." This is only the 42nd Mersenne prime found since Marin Mersenne, a 17th century French monk, first studied these numbers over 350 years ago. Written out, the number has 7,816,230 digits, over half a million digits larger than the previous largest known prime number. It was discovered after more than 50 days of calculations on a 2.4-GHz Pentium 4 computer.

Omniscient Debugging

An easier way to find program bugs

BIL LEWIS

The term "omniscient debugging" describes the concept that debuggers should know everything about the run of a program, that they should remember every state change, and be able to present to you the value of any variable at any point in time. Essentially, omniscient debugging means that you can go backwards in time.

Omniscient debugging eliminates the worst problems with breakpoint debuggers—no "guessing" where to put breakpoints, no "extra steps" to debugging, no "Whoops, I went too far," no nondeterministic problems. And when the bug is found, there is no doubt that it is indeed the bug. Once the bug occurs, you will never have to run the program a second time.

Breakpoint debugging is deduction based. You place breakpoints according to a heuristic analysis of the code, then look at the state of the program each time it breaks. With omniscient debugging, there is no deduction. You simply follow a trail of "bad" values back to their source.

In 1969, Bob Balzer implemented a version of omniscient debugging for Fortran that ran on a mainframe and had a TTY interface (see "EXDAMS—Extendible Debugging and Monitoring System," ACM Spring Joint Computer Conference, 1969). Since then, there have been a half dozen other related projects that were also short-lived. Zstep (Lieberman) was a Lisp debugger that came closest to commercial production and shared a similar GUI philosophy (see "Debugging and the Experience of Immediacy," by H. Lieberman, *Communications of the ACM*, 4, 1997). With the advent of Java (and its beautiful, simple, bytecode machine!), there have been at least five related projects, including two commercial products—CodeGuide from OmniCore (http://www.omnicore.com/) and RetroVue from VisiComp (http://www.visicomp.com/).

The ODB I present here implements this concept in Java. The ODB is GPL'd and available (with source code) at http://www.LambdaCS.com/. It comes as a single jar file that includes a manual, some aliases (or .bat files), and three demo programs. The current implementation is fairly stable, though not perfect. It is pure Java and has been tested (more or less) on Solaris, Mac OS, and Windows 2000. I am working on integration with Eclipse and IntelliJ. You are welcome to try it out and encouraged to let me know what you think.

Bil has written three books on multithreaded programming along with the GNU Emacs Lisp manual and numerous articles. He is currently a professor at Tufts University and can be contacted at bil@lambdacs.com.

The basic implementation of the ODB is straightforward. In Java, it is sufficient to add a method call before every *putfield* bytecode that records the variable being changed, its new value, the current thread, and the line of code where it occurred. (The ODB records more than this, but this is the fundamental idea.)

> "Omniscient debugging eliminates the worst problems with breakpoint debuggers"

With instrumentation taken care of, you can turn your attention to the real issues: How this information is going to be displayed and how you are going to navigate through time to find important events. All of this should help answer the question: "Does omniscient debugging work?"

Display and Navigation

The ODB GUI is based on a few fundamental principles:

- It displays all of the most relevant state information simultaneously. (You should not have to toggle between popups.)
- It provides easy-to-read, unambiguous print strings for objects. (*<Thing_14>* is a reasonable choice, *corp.Thing@0a12ae0c* isn't.)
- It makes the most frequent operations simple. (You shouldn't have to do text selections, use menus, and the like.)
- It doesn't surprise you. (Displays are consistent and you can always go back to where your were.)

The ODB records events in a strict order and assigns them timestamps. Every variable has a known value (possibly "not-yet-set") at every timestamp and everything in the debugger window is always updated to show the proper data every time you revert the debugger to a different time. There is never any ambiguity.

(continued from page 16)

You mainly want to know what the program state available to a chosen stack frame is—local variables, instance variables of the *this* object, and perhaps some static variables and details about other selected objects. You also need to know which line of code in which stack frame you are looking at. This is what the ODB displays. Such things as the build path, variable types, inheritance structures, and the like, are rarely of interest during debugging and shouldn't take up valuable screen space.

For print strings, the ODB uses the format *<Thing_34 Cat>*, where the type (without the package) is shown with an instance counter and an optional, programmer-chosen instance variable. An important detail is that the print string must be immutable—it has to be the same at time 0 as it is at time 10,000. Print strings have a maximum width of 20 characters, so that wrap-around isn't an issue. This format also lends itself well to method traces:

```
<Person_3 John>.groom(<Pet_44 Dog>, 44.95) -> true
```

In Figure 1, method calls are shown per-thread and indented, making clear the calling structure of the program. Traces are wonderful for orienting you to where/when in the program you are.

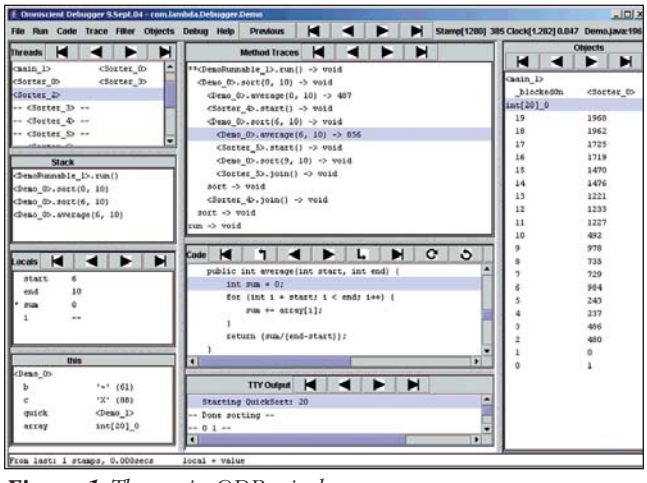


Figure 1: *The main ODB window.*

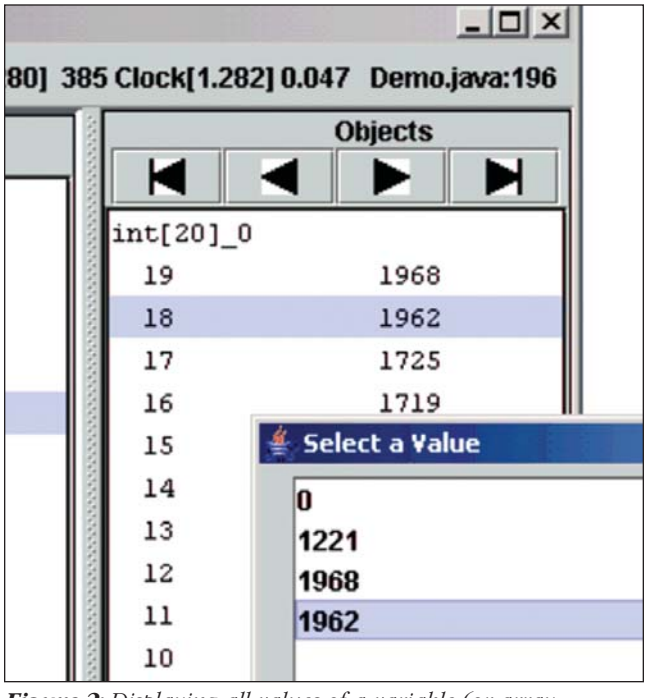


Figure 2: *Displaying all values of a variable (or array element).*

The primary operations I usually want are "show the previous/next value of a variable," and sometimes the first/last value. This is what the arrow buttons do for the selected line (for instance, next context switch, next variable value, next line of code, next line of output, and so on). Threads, traces, output lines, and lines of code may be selected directly, in which case the current time reverts to when that line was executed/printed. On occasion, it is useful to see the entire set of values a variable ever had (Figure 2). Being a rarer operation, it is on a menu.

At the bottom of the window is a minibuffer where messages are printed and where typed interactions occur. When you change the current time, the minibuffer displays the type of the newly selected timestamp (for example, the local assignment as in Figure 2) and how far it is from the previously selected time. Incremental text searches through the trace pane entries (based on EMACS's I-search) are run here, as are expression evaluation and more elaborate event searches.

In addition to the variables in the current stack frame, it is common to keep an eye on specific objects. Any object displayed in any pane may be copied to the Objects Pane where it remains as long as desired. As with all variables, the contents of those objects are updated as you revert the ODB to different times. Variables whose values have changed between the previous and currently selected time are marked with a "*" (*sum* in Figure 1). Those not yet set are displayed with a value of "–" (*i*). Similarly, thread *<Sorter_4>* has not yet executed its first instruction.

More Elaborate Interactions and Displays

There are times when more elaborate displays or interactions are useful. For example, when there are thousands (or millions!) of traces, it is useful to filter out uninteresting ones. You can filter the trace pane to show only those method calls in a particular package, or those not in that package, or only those at a depth less than a limit.

You can revert the debugger to any time desired, then evaluate a method call using the then current values of the recorded objects. These methods will be executed and their traces will be collected in a different timeline, which won't affect the original data. In the secondary timeline, it is possible to change the value of instance variables before executing a method. (The primary timeline is inviolate.) The input format is identical to the trace format and even does name completion.

There are some cases where a different print format is more intuitive for you. Thus, a blocked thread acquires a pseudovariable to show the object it's waiting on (see *_blockedOn* in Figure 1), as will the objects whose locks are used. Notice that this information is duplicated in the threads pane.

The collection classes constitute another special case. In reality, a *Vector* comprises an array of type *Object*, an internal index, and some additional bookkeeping variables that are not of any interest to anyone other than the implementer. So the ODB displays collections in formats more consistent with the way they are used.

In addition to the simple text search, there is an event search command that lets you look for specific assignments, calls, and the like (based on the work of M. Ducassi; see "Coca: An Automated Debugger for C" in the *Proceedings of the 21st International Conference on Software Engineering*, 1999). A good example of its use is the quick sort demo that is part of the ODB distribution. The program sorts everything perfectly, except for elements 0 and 1. To search for this problem, you can issue an event search to find all calls to *sort()* that include elements 0 and 1.

(continued from page 18)

```
port = call & callMethodName = "sort" & arg0 = 0 & arg1 >= 1
```

There are four matches, and *sort(0, 2)* is the obvious call to look at. Stepping through that method, it's easy to find the bug. Event searching has a long and honorable history of providing a "super intelligent" breakpoint facility. I find it a nice surprise that it's so useful here, too.

You can write ad hoc display methods for specific objects. For example, when debugging the ODB itself, I have a method that converts the bit fields of the timestamps (which are integers) into readable strings. Instead of seeing "5235365," I see "Thread-1 | a=b | Demo.java:44."

The Snake In the Grass

If you see a snake in the grass and you pull its tail, sooner or later you get to its head.

Under the ODB, a great number of bugs exhibit this kind of behavior. (By contrast, breakpoint debuggers suffer from the "lizard in the grass" problem. Even when you can see the lizard and grab its tail, the lizard breaks off its tail and gets away.) If a program prints something wrong ("The answer is 41" instead of 42), then you have a handle on the bug (you can grab the tail).

"Pulling the tail" consists of finding the improper value at each point, and following that value back to its source. It is not unusual to start the debugger, select the faulty output, and navigate to its source in 60 seconds. And you can do this with perfect confidence for bugs that take hours with conventional debuggers.

A good example of this is when I downloaded the Justice Java class verifier (http://jakarta.apache.org/bcel/). I found that it disallowed a legal instruction in a place where I needed it. Starting with the output "Illegal register store," it was possible to navigate through a dozen levels to the code that disallowed my instruction. I changed that code, recompiled, and confirmed success. This took 15 minutes (most of which was spent confirming the lack of side effects) in a complex code base of 100 files I had never seen before.

A happy side benefit of the ODB is how easy it makes it to see what a program is doing. I teach a class on multithreaded programming and I use the ODB regularly to explain what a program is doing. Students get to see which threads ran when

The ODB At Work

When I push zoom a second time, it displays the wrong data. Sometimes.

I had spent several hours looking for the bug with Eclipse. He was fairly sure that a certain *ArrayList* containing time-event pairs was being corrupted, but nothing more. He thought it would be interesting to try the ODB.

The EVolve visualization system comprises some 80K lines of Java and is designed to read-in and display performance data on Symbian applications for mobile phones. It had been written over several years by several programmers, none of whom were available. I had never used the tool, nor had I ever seen a single line of the source code. It was exactly what I was looking for!

The first indication of the problem was a dialog box that displayed a zero time. So I did an incremental search through the traces in the AWT thread for the string "Start time." Of the eight events, which contained that string, the second-to-last showed a zero time range.

I could see from the code that the string was constructed using the start and end instance variables from a Selection object. This particular object had the same value for both. Selecting that object and stepping back to its creation, I could see that *RefDim.makeSelection()* was calling *new* with those values. That method was called from *Paladin.select()*, which obtained those values by taking the difference between the start values of elements two and four of the aforementioned *ArrayList*. I noticed that the first five elements of the list were the same object.

Stepping backwards to find out who put those values in, I discovered this odd little loop, which ran in thread *T1*. (At this point, there is no clear connection between the creation of the list and its use.)

```
while ((x/Math.abs(interval) - xOffset) >=   timeMap.add(time2event);
}
```

It was clear that multiple identical entries in the list were allowed, but that these particular ones were wrong. After staring blindly at the loop for a while, I stepped back to the caller:

```
countEvents(x + xOffset*interval);
```

The programmer was adding the offset to the *X* value, only to remove it in the loop. Weird. I had noticed that the second selection only failed if it were in a low range (presumably *0 - xOffset*interval*, which was also the range where the *ArrayList* values were identical).

Removing the offset eliminated the bug. The entire session lasted about an hour, during which I looked at 20 objects in a dozen files. Most of my time was spent trying to understand the intent of the code.

—B.L.

(continued from page 20)
and if there's any confusion, they can just back up and look again. Some problems that used to be difficult suddenly become trivial.

Implementation

The ODB keeps a single array of timestamps. Each timestamp is a 32-bit *int* containing a thread index (8 bits), a source-line index (20 bits), and a type index (4 bits). An event for changing an instance variable value requires three words: a timestamp, the variable being changed, and the new value. An event for a method call requires a timestamp and a *TraceLine* object containing: the object, method name, arguments, and return value (or exception), along with housekeeping variables. This adds up to about 20 words. A matching *ReturnLine* is generated upon return, costing another 10 words.

Every variable has a *HistoryList* object associated with it that is just a list of timestamp/value pairs. When the ODB wants to know what the value of a variable was at time 102, it just grabs the value at the closest previous time. The *HistoryList* for local variables and arguments hang off the *TraceLine*; those for instance variables hang off a "shadow" object that resides in a hashtable.

Every time an event occurs, the recording method locks the entire debugger, a new timestamp is generated and inserted into the list, and associated structures are built. Return values and exceptions are back-patched into the appropriate *TraceLine* as they are generated.

Code insertion is simple. The source bytecode is scanned, and instrumentation code is inserted before every assignment and around every method call. A typical insertion looks like this:

```
289 aload 4     // new value
291 astore_1    // Local var X
292 ldc_w #404 <String"Micro4:Micro4.java:64">
295 ldc_w #416 <String "X">
298 aload_1     // X
299 aload_2     // parent TraceLine
300 invokestatic #14 <change(String, String, Object, Trace)>
```

where the original code was lines 289 and 291, assigning a value to the local variable *X*. The instrumentation creates an event that records that on source line 64 of *Micro.java* (line 292), the local variable *X* (line 295), whose *HistoryList* can be found on the *TraceLine* in register 2 (line 299), was assigned the value in register 1 (line 298). The other kinds of events are similar.

RetroVue At Work

Allen had finally decided that there was a bug in the JVM. Either that, or he was losing his mind. This was one of those bugs that "couldn't happen."

Allen used to like null pointer exceptions. The stack backtrace always identified the offending line, which usually indicated the offending variable, and then (eventually) he would find the bug. But lately he had started to dread the null pointer exception. "The stack backtrace helped me to understand the symptom of the bug, but the code containing the actual cause of the bug was often far removed from the exception throwing code," he explained. "And by the time the exception was thrown, the underlying cause was sometime in the past. Who assigned null to the variable? And when? Where? That's the real bug, and by the time the exception occurs, it's usually way too late to pinpoint the cause."

In this particular case, the null value was contained in a field, so it wasn't too difficult using conventional tools to find all the places in the code where it was used. From that list, Allen whittled it down to a long list of assignments.

But upon examination of the code, it seemed none of them could be the cause. In each case, if the value were null, it would have blown up a few lines before the assignment, because in each case the value was dereferenced. Allen dutifully added some print statements, but they just confirmed what he had already surmised. He examined the constructors and added code to check the initialization values, but they, too, always verified that the value being assigned was nonnull.

After two days of tearing his hair out, he e-mailed his colleague Carl for help. "I don't think Allen really expected me to be able to help," Carl said. "I work remotely, and I wasn't familiar with Allen's code. But he sounded desperate." Carl asked Allen to run the program under RetroVue and send him the resulting journal file. Ten minutes after receiving it, Carl sent Allen information (see Figure 3) that he had captured from the RetroVue journal viewer and marked up. RetroVue clearly showed that, although the value of the argument was not null (line 115), the field named "configuration" was indeed being assigned a null value in *GraphElement*'s constructor on line 119. That would explain the eventual null pointer exception. But why wasn't the argument value being assigned to the field? Allen examined the constructor one more time:

```
115 public GraphElement(GraphNode parent, Configuration configuration) {
116 if (configuration == null) {throw new IllegalArgumentException();}
117 this.parent = parent;
118 this.children = new GraphNode[0];
119 this.configuration = configuration;
120 }
```

Finally, he noticed the spelling error. Because the configuration argument was missing a "u," it was never used in the constructor at all (except in the check for null, which Allen had added, inadvertently duplicating the spelling error with copy-and-paste). So line 119, which was supposed to copy the nonnull argument value into the field, simply copied the field to itself (as if the assignment statement had been written *this.configuration = this.configuration;*). And, of course, the field's initial value was null.

The incident convinced Allen to start using RetroVue himself. "We had found and fixed the bug, but now I was curious why it occurred intermittently. Using RetroVue, I got a clear understanding of exactly what was going on in these classes in less than an hour. And along the way, I found and fixed another potential bug, and discovered the cause of a performance problem that another engineer had been struggling with."

— Ron Hughes
http://www.visicomp.com/

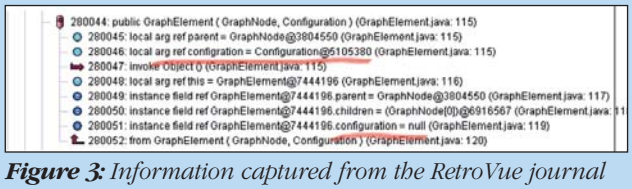


Figure 3: *Information captured from the RetroVue journal viewer.*

CodeGuide At Work

As the producers of an IDE that includes omniscient debugging facilities, naturally we use back-in-time debugging daily during development. CodeGuide works a bit differently than ODB in that it marries conventional debugging with back-in-time facilities. Thus, the developer can use conventional breakpoints when he wants to and go back in time when necessary. Once a breakpoint is set, CodeGuide instructs the debugged application to log execution details in the method containing the breakpoint and "related" methods. The rest of the application is not affected and can continue to run at full speed. If the bug is easy to fix, the developer can HotSwap his changes into the VM and test them without the need to restart the application.

Bugs in Java applications often manifest themselves in uncaught exceptions. The dreaded *NullPointerException* is probably the most common example.

Now, once a *NullPointerException* is thrown because the return value of a function is null, you usually have no clue why this function returned null. For example, I recently had to find a bug in some code that uses a cache to hold some per-file data. With omniscient debugging in CodeGuide, I just had to set a breakpoint on the main exception handler of this thread, and then I stepped backwards to the cause of the exception.

The code in which the *NullPointerException* was thrown looked like this:

```
class Processor {
   private Cache cache;
   public String process(File file) {
//   ...
Data data = cache.getData(file);
//   ...
return data.getStringRepresentation();
// <- NPE thrown
   }
}
```

It was clear that the *Cache.getData()* method was at fault for returning null. But how could that happen? The *Cache.getData()* method was not supposed to ever return null. It was supposed to return dummy data instead:

```
class Cache {
  private Map<File, Data> cachedFiles = new HashMap<File, Data>();
private Data getData(File file) {
  if (!cachedFiles.containsKey(file))
    return new DummyData();
return cachedFiles.get(file);
  }
}
```

Stepping back into the *Cache.getData()* method revealed that the *cachedFiles.get(file)* returned null, even though *cachedFiles.containsKey(file)* returned True. A peculiarity of *HashMap* is that it allows storing null values, in contrast to *Hashtable,* which does not. Thus, changing the code to not use *HashMap.containsKey()* fixed the problem:

```
private Data getData(File file) {
   Data data = cachedFiles.get(file);
   if (data == null)
return new DummyData();
  else
return data;
}
```

Could I have found this bug without omniscient debugging? Certainly. But it would have taken much, much longer and several debugging sessions with breakpoints in different locations to get to the root of this.

— Hans Kratz
http://www.omnicore.com/

(continued from page 22)

Performance

The most common remarks I hear when introducing omniscient debugging are that it takes too much memory and it is much too slow. These are legitimate concerns, but miss the main point— does it work? If it is effective, there are all sorts of clever techniques for reducing the cost of the monitoring. If it isn't, who cares?

In experience with the ODB, neither CPU overhead nor memory requirements have been a stumbling block. On a 700-MHz iBook, it takes the ODB 1µs to record an assignment. I have recorded 100 million events in a 31-bit address space. A tight loop may run 300 times slower in a naïve ODB, but such a loop can also be optimized to do no recording at all (it's recomputable). Ant recompiling itself runs 7 times slower and generates about 10 million events. (We're recording Ant, not the compiler!)

There are a number of possible optimizations:

- You can start/stop recording either by hand or by writing an event pattern. Most button-push bugs are amenable to this. (Let the program start up with recording off, turn it on, push the button, turn it off, find the bug.)
- The ODB has a garbage collector that throws out old events as memory gets filled up. Of course, these are not really garbage, but it does make it possible to maintain a sliding window of events around the bug. (I find this is a grand idea, but only seldomly useful.)
- It is quite normal for a large percentage of a program to consist of well-known, "safe" code. (These are either recomputable methods, or methods we just don't want to see the interiors of.) By requesting that these methods not be instrumented, a great number of uninteresting events can be eliminated. The ODB lets you select arbitrary sets of methods, classes, or packages, which can then be instrumented or not. A good optimizer can do this automatically.

My estimate is that a well-optimized omniscient debugger would exhibit an absolute worst-case of 10 times slow-down, with 2 times being far more common. True, there are programs for which this is still too long, but not many. I've yet to encountered a bug that I couldn't find with the current ODB.

Conclusion

With the advent of omniscient debugging, it is becoming easier to find bugs. Bugs that were completely intractable with traditional techniques have become child's play. As optimization techniques improve and omniscient debuggers come to handle larger and larger problems, bugs will find it harder and harder to hide.

DDJ

Examining Software Testing Tools

Class-based testing goes to work

DAVID C. CROWTHER
AND PETER J. CLARKE

In a world where automatic updates and the latest service packs are quick fixes for common problems, the need for higher quality control in the software-development process is evident and the need for better testing tools to simplify and automate the testing process apparent. Why are so many developers seeking automated software testing tools? For one reason, an estimated 50 percent of the software development budget for some software projects is spent on testing [16]. Finding the right tool, however, can be a challenge. How is one tool measured against another? What testing features are important for a given software development environment? Does a testing tool really live up to its claims?

In this article, we analyze several software testing tools—JUnit, Jtest, Panorama for Java, csUnit, HarnessIt, and Clover.Net. The first three tools were developed to work with Java programs, while the latter three work on C# programs. We chose tools that ranged from freeware to commercial, and from basic to sophisticated. Our focus in selecting the tools—as well as the tests we performed—was on class-based testing. Our intention here is not to provide a comprehensive comparison of a set of testing tools, but to summarize the work completed in an independent study in an academic environment.

David is a graduate student and Peter an assistant professor in the school of computer science at Florida International University. They can be contacted at dccrowther@alumni.cs.fiu.edu and clarkep@cs.fiu.edu, respectively.

Software Testing Overview

Binder defines software testing as the execution of code using combinations of input and state selected to reveal bugs. Its role is limited purely to identifying bugs [2], not diagnosing or correcting them (debugging). The test input normally comes from test cases, which specify the state of the code being tested and its environment, the test inputs or conditions, and the expected results. A test suite is a collection of test cases, typically related by a testing goal or implementation dependency. A test run is an execution of a test suite with its results. Coverage is the percentage of elements required by a test strategy that have been traversed by a given test suite. Regression testing occurs when tests are rerun to ensure that the system does not regress after a change [3]. In other words, the system passes all the tests it did before the change.

Components usually need to interact with other components in a piece of software. As a result, it is common practice to create a partial component to mimic a required component. Two such instances of this are:

- Test driver, a class or utility program that applies test cases to a component to be tested.
- Test stub, a partial, temporary implementation of a component, which lets a dependent component be tested.

> "Software testing is the execution of code using combinations of input and state selected to reveal bugs"

Additionally, a system of test drivers and other tools to support test execution is known as a "test harness," and an "oracle" is a mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test [8].

Class-based testing is the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of some aspect of the class [4]. This definition is based on the IEEE/ANSI definition of software testing [5]. Class-based testing is comparable

(continued from page 26)
to unit testing used for procedural programs, where each class is tested individually against its specification. The two main types of class-based testing are specification based and implementation based.

Specification-based testing, also known as black box or functional testing, focuses on the input/output behavior or functionality of a component [3]. Some methods used for specification-based testing include equivalence, boundary, and state-based testing. In equivalence testing, values in the domain are partitioned into equivalence classes, in which any value tested should produce the same result as any other value in that class. Boundary testing focuses on testing the extreme input values, such as the minimum and maximum values along with other values within their proximity. Finally, state-based testing generates test cases from a finite state machine (a statechart, for instance), which models some functionality of the system [3, 7].

Implementation-based testing, also known as white box or structural testing, focuses purely on the internal structure of a coded component [3]. The most common implementation-based techniques are based on data flow and control flow analysis of code, which generate test cases based on some coverage criteria [2]. In data flow analysis, each variable definition is matched up with the places in code where the variable is used, constituting a *def-use* pair. These pairs are used for measuring coverage; for instance, "All-Uses" checks if there is a path from every *def* to each use of a variable [6]. Control flow testing analyzes coverage based on the order in which statements are executed in the code. For example, branch testing seeks to ensure every outcome of a conditional statement is executed during testing, and statement coverage verifies the number of statements executed.

While both specification-based and implementation-based testing have their advantages and disadvantages, it is generally accepted that some combination of the techniques is most effective [1].

Criteria for Comparison

Here are the criteria we used to evaluate the selected software testing tools:

Support for the testing process. How does a testing tool support the actual testing process? For each of the following items, we note whether they are supported by each of the testing tools. Possible result values for each criterion are: Y (for Yes), N (No), and P (Partial).

1. Support for creation of stubs/drivers/harnesses.
   a. Does the testing tool provide a mechanism for generating stubs?
   b. Does the testing tool provide a mechanism for generating drivers?
   c. Does the testing tool provide a mechanism for generating test harnesses?
2. Support for comparing test results with an oracle.
- Does the tool provide a mechanism for automatically comparing the results of the test against the expected results?
3. Support for documenting test cases.
- Does the tool assist in keeping documentation of test cases?
4. Support for regression testing.
- Does the tool provide the ability for automated regression testing?

Generalization of test information. What additional information can be provided from a testing tool, to provide further support for the testing process?

1. Support for the creation of test cases.
   a. Does the tool provide help with creating test cases?
   b. Does the tool provide help with the generation of implementation-based test cases?
   c. Does the tool provide help with the generation of specification-based test cases?
2. Support for structural coverage.
   a. Does the tool provide a measure of statement coverage?
   b. Does the tool provide measure of branch coverage?
   c. Does the tool provide a measure of all-uses coverage?

Usability of testing tool. How easily can a given tool be installed, configured, and used? As these criteria are more subjective, these factors are rated for each of the selected tools. Each element listed here is given a ranking from 1 to 5, where 5 is always the positive or more desired result, 3 is an average or ordinary result, and 1 indicates a poor or very negative result. For example, "Learning Curve" is based on the time needed to become familiar with the tool before being able to use it in practice, and a shorter amount of time results in a higher score.

1. Ease of Use
   a. How easy is it to install the tool?
   b. How user friendly is the interface?
   c. What is the quality of online help?
   d. How helpful are the error messages?
2. Learning Curve
- How long should it take an average programmer to be able to use the tool efficiently in practice?
3. Support
   a. How quickly can you receive technical support information?
   b. How helpful is the technical support provided?

Requirements for testing tool. What requirements and capabilities of the tools help determine which one(s) might be suitable for your circumstances?

1. Programming Language(s).
- What programming language(s) will the tool work with?
2. Commercial Licensing.
- What licensing options are available and at what costs?
3. Type of testing environment.
- Is the testing environment command prompt, Windows GUI, or part of the development environment?
4. System Requirements.
   a. What operating systems will the tool run on?
   b. What software requirements does the tool necessitate?
   c. What hardware requirements does the tool impose?

Testing Tools Analyzed

JUnit is a freely available regression testing framework for testing Java programs, written by Erich Gamma and Kent Beck (http://www.junit.org/) [9]. It allows automated unit testing of Java code and provides output to both text-based and Windows-based user interfaces. To set up test cases, you write a class that inherits from a provided test case class, and code test cases in the format specified. The test cases can then be added to test suites where they can be run in batches. Once the test cases and test suites have been configured, they can be run again and again (regression testing) with ease. JUnit runs on any platform, needing just the Java runtime environment and SDK installed.

Jtest is a commercially available automated error-prevention tool from Parasoft (http://www.parasoft.com/). It automates Java unit testing using the JUnit framework, as well as checks code against various coding standards [11]. After a Java project is imported into Jtest, Jtest analyzes the classes, then generates and executes JUnit test cases. The generated test cases attempt to maximize coverage, expose uncaught runtime exceptions, and verify requirements. Additional test cases can be added to those produced by Jtest for more specific testing criteria. Setting Jtest apart from other testing tools is its ability to check and help correct code against numerous coding standards.

Created by International Software Automation (ISA), Panorama for Java is a fully integrated commercially available

software engineering tool that assists you with software testing, quality assurance, maintenance, and reengineering [12] (http://www.softwareautomation.com/java/index.htm). Panorama provides testing analysis, including code branch test coverage analysis, method and branch execution frequency analysis, test case efficiency analysis, and test case minimization. Panorama provides a GUI capture/playback tool to let you execute test cases manually using the application's interface and then later play them back again. In addition to its testing capabilities, Panorama generates various charts, diagrams, and documents, which help measure software quality and make complex programs easier to understand.

csUnit is a freely available unit testing framework for .NET, which allows testing of any .NET language conforming to the CLR, including C#, VB.Net, J#, and managed C++ [13] (http://www.csunit.org/). Test cases are written using the functionality provided by the framework, and the test cases and test suite are identified by adding attributes to the corresponding functions and class. To run the test cases, you launch *csUnitRunner* and point it at the project's binary file; then the GUI displays the results of the test.

HarnessIt, developed by United Binaries, is another commercial unit testing framework for .NET (http://www.unittesting.com/). Like csUnit, it supports any .NET [14] CLR-compliant language, and provides documentation on testing legacy code, such as unmanaged C++ classes. Test cases are written in the .NET language of choice using the provided framework and marked with the appropriate attribute. A GUI runs the test cases and displays the results. The interface can be run outside of Visual Studio or integrated to automatically run every time the program is debugged. While HarnessIt provides a similar functionality to csUnit and other unit-testing software, as a commercial tool it has superior documentation, features, and flexibility, and a full support staff for assistance.

Clover.Net is a code coverage analysis tool for .NET from Cenqua (http://www.cenqua.com/clover.net/). As such, it highlights code that isn't being adequately tested by unit test cases [15]. Clover.Net provides method, branch, and statement coverage for projects, namespaces, files, and classes, though it currently only supports the C# programming language. A Visual Studio .NET plug-in enables configuring, building, and displaying results of test coverage data. Clover can be used in conjunction with other unit testing tools, such as csUnit or HarnessIt, or with handwritten test code.

Comparative Study

The example class we selected to evaluate the testing tools was a stack, which was simple enough to make analyzing the tool output straightforward. At the same time, it had enough functionality to provide various places to "break" the code. We wrote the stack class in Java (Listing One) and C# (Listing Two), to accommodate testing tools of both languages. The code for the two languages was similar, differing only in which access modifiers were needed, case of built-in object operations, and the fact that the class was stored in a package in Java and a namespace in C#.

Each testing tool had its own unique features and functions for testing; nevertheless, all tools had in common the test case. The test case is the most basic unit involved in testing, yet all other aspects of testing depend on it. Some tools require you to write the code for the test cases, while other tools generate test cases for you automatically.

JUnit. To set up test cases in JUnit, you write a test class, which inherits from the JUnit *TestCase* class. Each function in the class represents a test case where the code creates one or more instances of the class to be tested and executes functions of the class using parameters as necessitated by the test case. Listing Three is a test case that tests the *pop* function of *Stack*. Here, two stacks are created: one to test with (*stTested*) and the other to hold the expected result (*stExpected*). After the object being tested has been exercised appropriately, one or more *assert* functions are executed to check if the expected result(s) matches the actual result(s). In this example, one assert tests that the correct value is being popped off, then another assert checks that the final object matches the expected object. The JUnit framework catches any exceptions that are thrown by the asserts and handles them accordingly.

Test cases are added to test suites where they can be executed in sets, or, if all test case names begin with "test," the test runner automatically creates a test suite with all the test cases and runs them. JUnit provides both text and GUI interfaces for the test runner. Configuration of the test cases takes time, but the core functionality of JUnit is regression testing, where a configured test case can be reexecuted any number of times.

Jtest. Jtest automates the test case generation process by analyzing each class and creating test cases for each function of the class. These test cases attempt to maximize coverage, expose uncaught runtime exceptions, and verify requirements by using the JUnit framework. Listing Four is a sample test case generated by Jtest where a test stack is created from an empty object array and an object is pushed onto it. The test case then verifies that the resulting *Stack* is not null. There are many similar white box test cases generated by Jtest, which attempt to execute as much of the code as possible; however, black box testing is not necessarily as automated.

For Jtest to automate black box testing, the specifications must be embedded into the code using Object Constraint Language (OCL), a language embedded into code as comments, which allows a class's constraints to be formally specified [3]. Otherwise, you can either modify the automatically generated test cases to test the specification, or write JUnit test cases manually outside of the Jtest-generated cases. Regression testing is performed by reusing code similar to JUnit.

There are many additional functionalities Jtest provides to enhance the testing process; for example, it automatically creates stub and driver code for running test cases. Also, Jtest measures code coverage to show completeness of test cases. Finally, it checks code against numerous coding standards and can automatically correct many such issues. A testing configuration is completely customizable to include or exclude any of the numerous features/subfeatures.

Panorama for Java. Creating unit test cases like those just described is not possible in Panorama. The way to create test cases in Panorama is to manually execute them in the application's interface while they are recorded. These test cases could later be replayed where Panorama would reexecute the same test cases using the application's interface, and thereby perform regression testing. However, this tests the user interface, not the class itself.

csUnit. In csUnit, test code is marked with attributes that allow the framework to identify the various test components. Test cases are placed in a test class, which is preceded by a *[TextFixture]* attribute, while the test cases themselves are preceded by a *[Test]* attribute. An *Assert* class is provided by the framework to validate whether a test case passed. Listing Five is the *testPopEmpty* test case. Here, *Stack st* is instantiated as an empty stack. An *Assert.True* is then used to test that an attempted pop would return null.

Test cases are run by the csUnitRunner application, which can be started from the Program menu under csUnit or by choosing csUnitRunner from the Tools menu in Visual Studio. In csUnitRunner, test cases are loaded by choosing File|Assembly|Add and pointing to the binary file produced by building the solution to test. Then, clicking the Run icon or choosing Test|Run All starts csUnitRunner, executing each test case and displaying the subsequent results.

HarnessIt. HarnessIt test cases, like csUnits, are identified with an attribute; only in this case, the attribute is *[TestMethod]*, and the class containing it is noted with a *[TestClass]* attribute. The method of evaluating a test case, though, is different. Instead of using an *Assert* class to test validity, a *TestMethodRecord* type is provided, and is passed to a test case function. After the test case has been set up, the *RunTest* method of the *TestMethodRecord* type is called and passed a Boolean statement which can include unit test commands, comparison operators, and expected results. Listing Six is the test case for *testPushFull*, which fills *Stack s1* with the maximum number of values. *Stack s2* is instantiated as a duplicate of *Stack s1*. Then the code attempts to push another value onto *s1*. The *TestMethodRecord* verifies that *s1* still equals *s2*, because no more values can be pushed onto a full stack.

Test files are run from the HarnessIt application, which can be opened from the HarnessIt option in the Program menu. Inside HarnessIt, choosing File|Add Assembly and pointing to the project's binary loads the project for testing. The test cases can then be run by choosing File|Run from the menu or pressing F5. A window pops up with an indicator bar showing progress as the tests are being run. The indicator bar remains green as long as tests pass, and turns red when they fail. Once finished, a report is generated, displaying all test cases in a hierarchical view with their test execution status.

Clover.Net. Clover.Net, as a coverage analysis tool, does not provide a mechanism for creating test cases, but rather, assumes test cases have already been coded. Clover.Net analyzes the test cases when they are executed, to produce coverage results. Clover.Net provides a Visual Studio plug-in, which lets you configure testing options as well as view coverage results directly in Visual Studio. Clover can even highlight the code that hasn't been tested, making it easy to find which areas were missed. In the Clover View, functions are marked green if they have been fully exercised, yellow if they have been partially exercised, and red if they have not been exercised at all. Statistics are given for the percentage of methods, branches and statements covered, and these statistics can be broken down from the solution level to project, class, or even function level.

Criteria/Tools	JUnit	Jtest	Panorama for Java	csUnit	HarnessIt	Clover.Net
1a.Creation of Drivers	P	Y	N	P	P	N
1b. Creation of Stubs	N	Y	N	N	N	N
1c. Creation of Test Harness	Y	Y	Y	Y	Y	Y
2. Compare Results with Oracle	Y	Y	N	Y	Y	N
3. Help with Documenting Test Cases	N	N	N	N	N	N
4. Capable of Regression Testing	Y	Y	Y	Y	Y	Y

Table 1: *Support for the testing process.*

Criteria/Tools	JUnit	Jtest	Panorama for Java	csUnit	HarnessIt	Clover.Net
1a. Creation of Test Cases	N	Y	N	N	N	N
1b. Implementation-based Test Cases	N	Y	N	N	N	N
1c. Specification-based Test Cases	N	Y	N	N	N	N
2a. Branch Coverage	N	Y	Y	N	N	Y
2b. Statement Coverage	N	Y	Y	N	N	Y
2c. All-uses Coverage	N	N	N	N	N	N

Table 2: *Generalization of test information.*

Criteria/Tools	JUnit	Jtest	Panorama for Java	csUnit	HarnessIt	Clover.Net
1a. Ease of Installation	3.9	4.4	4.3	5	4.9	3.4
1b. User-friendliness of Interface	3.7	4.4	4.3	5	4	4
1c. Quality of Online Help	3.8	3.4	3.3	3.1	4.7	4.2
1d. Helpfulness of Error Messages	4.3	4.23.	5	3.9	4.3	4.2
2. Learning Curve	3.9	3.4	3.8	4.6	3.8	3.2
3a. Responsiveness of Technical Support	N/A	3.5	3	N/A	5	4.7
3b. Helpfulness of Technical Support	N/A	3.5	3	N/A	5	4.7

Table 3: *Usability of testing tool.*

Test Results

Each tool was examined according to the criteria presented here. Tables 1 through 4 detail the results of these evaluations, which correspond to the four sets of criteria. Tables 1 and 2 deal with features that a tool may or may not have. The result cells for each tool contain a *Y* (Yes, it has this feature), *N* (No, it doesn't support this), or *P* (Partial functionality of the feature available). Table 3 presents more subjective criteria, for which a score of 1 to 5 is assessed, where 5 = Excellent, 4 = Good, 3 = Average, 2 = Sub par, and 1 = Poor. Finally Table 4 provides additional information on requirements of the tools.

Support for the testing process. These criteria deal with the additional support a tool can provide to make the testing process easier (see Table 1). All of the tools provided a test harness for running the test cases, and the ability for regression testing. Many of the tools scored a "P" in the Creation of Drivers category because, though they provided a framework for creating drivers, Jtest was the only tool that actually automated this process, along with the creation of stubs. All the unit testing tools gave the ability to check results with a user supplied oracle. None of the tools provided help for documenting test cases, although some of the commercial tools may have an additional module for purchase to assist with this.

Generalization of test information. The criteria in this section deal with how well a tool supports the actual testing process, including automatic generation of test cases and various coverage measurements (Table 2). Not surprisingly, the freeware tools did not offer help in these areas, leaving it to you to perform these tasks manually. Jtest again scored well with the ability to automatically generate test cases, and provide branch and statement coverage results when running them. Panorama and Clover.Net both provided branch and statement coverage for test cases that were run.

Usability of testing tool. These criteria deal with ease of setup and use (Table 3). While the freeware tools scored well in these categories, bear in mind that they don't offer the functionality of the other tools. To substantiate the results and eliminate bias, the tools were distributed to a group of computer science students at Florida International University who ranked the tools on the preselected criteria.

Requirements of testing tool. Table 4 provides additional information on the requirements of each tool. In addition to the fact that several of the products can be used for multiple programming languages, others have similar products available for additional languages. The remaining requirements can help determine which tools would fit a particular environment, and whether hardware and software might need to be upgraded or purchased to use a given tool.

Analysis of Tools

JUnit provided a framework for creating, executing, and rerunning test cases. The documentation was easy to follow and gave thorough instructions on how to configure JUnit to work with existing code and set up test cases accordingly. The interface clearly showed which test cases failed and where, making it easy to find and correct problems. Once test cases had been set up, they could be executed over and over again for regression testing. Like other freely available tools, JUnit does not possess the automated aspects of tools, such as Jtest and Panorama.

JUnit provides a good starting point for testing Java programs. As a free program, there is an obvious cost benefit over more sophisticated tools. Additionally, the straightforwardness of setting up test cases makes the learning curve minimal. Many programmers find this an adequate testing tool; those working on larger applications would most likely seek more advanced tools.

Jtest took the JUnit framework and added automation to it. Jtest was a bit more difficult to set up than JUnit, and we had a hard time getting it configured properly with the documentation provided. On the other hand, Parasoft's technical support was very responsive, usually replying to e-mail within a day. We were given a direct number to a support agent, who remotely logged into our test computer and showed us how to configure Jtest to work with our code.

Jtest automated the process of creating test cases and stubs for our sample code, and provided an environment for running tests. Automation of white box test cases went seamlessly; however, for black box test cases, a little more work was required up front to insert OCL statements into the code. One advantage of doing so is that it produced code documentation along the way. In addition to testing functionality, the major selling point to this tool was the extensive ability to check code versus numerous different coding standards, and help bring the code into conformance.

Jtest is a full-featured advanced testing tool, and as such, comes with a price tag. It is intended for advanced users in a sophisticated software development environment. It provides lots of functionality to assist testers and programmers in developing quality products.

Panorama for Java was simple to install and go through using the sample program provided by Panorama. However, we were unable to get Panorama to run test cases for our sample class. This was because Panorama does not actually generate test cases, or provide a framework for creating them. What it does provide is the ability to record test cases that are manually executed, so that they can be played over again. This, however, is really used for testing the user interface, where we were looking to run test cases against the class itself in the code.

Panorama does provide sophisticated reports and graphs for measuring testing and the code itself. Perhaps in the future, Panorama will incorporate more capabilities for unit testing into its software suite.

csUnit provided a unit testing framework for .NET, in our case, a C# testing solution. The documentation was simple and easy to follow, though not overly extensive. The csUnitRunner interface was well designed with a hierarchical view of the test cases available to navigate, and the standard color-coded results indicating if a test passed or failed. Regression testing was achieved in the same manner as the other unit testing tools.

As a freeware tool, csUnit is appealing for budget-conscious and novice testers. Its simplicity is another quality to make it a good starting point for novices, as well as an acceptable test tool for time-constrained developers.

HarnessIt offered a testing product with similar functionality to csUnit at a nominal price. It supplied a full help menu which specified the basics of how to test code with the provided framework, as well as more advanced topics, such as using HarnessIt to test legacy code. HarnessIt provided the standard unit testing functionality, and displayed test results in an easy to follow manner. HarnessIt can be integrated into an application such that whenever you run the application in debug mode, it automatically launches and runs the test cases.

Overall, we found that HarnessIt did not provide much more functionality than csUnit; however, this does not necessarily mean it is not worth the price. What you are paying for is organized, complete help documentation, as well as knowledgeable support staff to help you in getting the tool working with your code.

Criteria/Tools	JUnit	Jtest	Panorama for Java	csUnit	HarnessIt	Clover .Net
1. Programming Language(s)	Java	Java (Parasoft does offer similar products for C/C++ and .NET)	Java (ISA does offer equivalents for C/C++ and VB)	Any .NET CLR Language	Any .NET CLR Language	C# (Cenqua provides a similar product for Java
2. Licensing	Open Software	See web site	See web site	Open Software	See web site	See web site
3. Type of Environment	Integrates into existing Java development environment	Command prompt, GUI, or could be integrated to certain development environments	GUI	GUI	GUI	GUI or command prompt
4a. Platforms	Any	Windows 2000 Windows Xp, Windows 2003 Server, Solaris Linux	Windows 95/98/NT	Windows 2000 Pro, Windows 2000 Advanced Server, Windows XP Pro	Windows	Windows
4b. Software Requirements	Current JDK and JRE	Sun Microsystems JRE1.3 or higher	Current JDK and JRE	Microsoft .NET Framework 1.0 or 1.1	Microsoft .NET Framework 1.0 or SP2 or higher	Microsoft .NET Framework 1.1 or Visual Studio,
4c. Hardware Requirements	None given	Intel Pentium III 1.0 GHZ or higher recommended SVGA (800x600 min.) (1024x768 rec.) 512 MB RAM min. 1 GB RAM rec.	At least 5MB free hard drive space. At least 16 MB RAM	None given	None given	None given

Table 4: *Testing tool requirements.*

We were able to integrate the Clover.Net tool into Visual Studio without much hassle by following the instructions in the Clover.Net manual. The manual was well laid out and contained all the necessary documentation for working with Clover. The Clover View provided coverage results within Visual Studio making it simple to find which areas of the code needed further testing. Clover.Net does not include a framework for creating test cases, but works in conjunction with unit testing tools such as the those mentioned previously.

As a commercial tool, Clover.Net offers significant additional functionalities over the available freeware tools by providing coverage analysis. Additionally, the price tag isn't much higher than HarnessIt. This tool would be ideal for a small to medium development environment that does not have the necessary budget or technical need of a tool on Parasoft's level.

Conclusion

We have examined here just a few of the available testing tools.

As the functionality of software expands, and the underlying code becomes more complex, the often neglected area of software testing will be called on more than ever to ensure reliability.

Acknowledgments

Thanks to the students in the Software Testing class at Florida International University, Fall 2004, for their participation in evaluating the tools. The members of the Software Testing Research Group also provided valuable comments that were incorporated in the article.

References

[1] J.D. McGregor and D.A. Sykes. *A Practical Guide To Testing Object-Oriented Software.* Addison-Wesley, 2001.

[2] R.V. Binder. *Testing Object-Oriented Systems.* Addison-Wesley, 2000.

[3] B. Bruegge and A.H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* Pearson Prentice Hall, 2004.

[4] P.J. Clarke. "A Taxonomy of Classes to Support Integration Testing and the Mapping of Implementation-Based Testing Techniques to Classes." Ph.D. Thesis for Clemson University, August 2003.

[5] IEEE/ANSI Standards Committee, 1990. Std 610.12.1990.

[6] M.J. Harrold and G. Rothermel. "Performing Data Flow Testing on Classes," *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM, December 1994.

[7] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y.Y. Toyoshima, C Chen, and J. Gao. "Object State Testing and Fault Analysis for Reliable Software Systems," *Proceedings of the 7th International Symposium on Reliability Engineering*, pages 239–242. IEEE, August 1996.

[8] BCS SIGIST. "Glossary of Terms used in Software Testing," http://www.testingstandards.co.uk/Gloss6_2.htm, June 2004.

[9] E. Gamma and K. Beck. JUnit. http://www.junit.org, June 2004.

[10] Parasoft. Automating and Improving Java Unit Testing: Using Jtest with JUnit. http://www.ddj.com/documents/s=1566/ddj1027024311022/parasoft.pdf.

[11] Parasoft. Jtest. http://www.parasoft.com, July 2004.

[12] International Software Association. Panorama for Java. http://www.softwareautomation.com/java/index.htm, July 2004.

[13] Manfred Lange. csUnit. http://www.csunit.org, August 2004.

[14] United Binary, LLC. HarnessIt. http://www.unittesting.com/, August 2004.

[15] Cenqua. Clover.Net. http://www.cenqua.com/clover.net/, August 2004.

[16] I. Sommerville. *Software Engineering.* Addison-Wesley, 2004.

DDJ

Listing One

```
//Simple stack implementation - Java
public class Stack
{
    protected static final int STACK_EMPTY = -1;
    protected static final int STACK_MAX = 1000;
    Object[] stackelements;
    int topelement = STACK_EMPTY;

    //create empty stack
    Stack()
    {
        stackelements = new Object[STACK_MAX];
    }
    //create stack with the objects stored in the array from bottom up
    Stack(Object[] o)
    {
        stackelements = new Object[STACK_MAX];
        for(int i=0;i<o. length;i++)
        {
            push(o[i]);
        }
    }
    //create stack as a duplicate of given stack
    Stack(Stack s)
    {
        stackelements = s. stackelements;
        topelement = s. topelement;
    }
    //push object onto the stack
    void push(Object o)
    {
        if(!isFull())
        {
            stackelements[++topelement] = o;
        }
    }
    //pop element off the stack
    Object pop()
    {
        if(isEmpty())
            return null;
       else return stackelements[topelement--];
    }
//return true if stack is empty
    boolean isEmpty()
    {
        if(topelement == STACK_EMPTY)
            return true;
        else return false;
    }
//return true if stack is full
    boolean isFull()
    {
        if(topelement == STACK_MAX-1)
            return true;
        else return false;
    }
}
```

Listing Two

```
//Simple stack implementation - C#
public class Stack
{
    public const int STACK_EMPTY = -1;
    public const int STACK_MAX = 100;
    private Object[] stackelements;
    private int topelement = STACK_EMPTY;

    //create empty stack
    public Stack()
    {
        stackelements = new Object[STACK_MAX];
    }
    //create stack with the objects stored in the array from bottom up
    public Stack(Object[] o)
    {
        stackelements = new Object[STACK_MAX];
        topelement = o. Length-1;
        for(int i=0;i<o. Length;i++)
        {
            stackelements[i] = o[i];
        }
    }
//create stack as a duplicate of given stack
    public Stack(Stack s)
    {
        stackelements = s. stackelements;
        topelement = s. topelement;
    }
//push object onto the stack
    public void push(Object o)
    {
        if(!isFull())
        {
            stackelements[++topelement] = o;
        }
    }
//pop element off the stack
    public Object pop()
    {
        if(isEmpty())
            return null;
        else return stackelements[topelement--];
    }
    //return true if stack is empty
    private bool isEmpty()
    {
        if(topelement == STACK_EMPTY)
            return true;
        else return false;
    }
//return true if stack is full
    private bool isFull()
    {
        if(topelement == STACK_MAX-1)
            return true;
        else return false;
    }
}
```

Listing Three

```
public void testSimplePop()
{
    String strA = "a";
    String[] st = new String[1];
    st[0] = strA;
    Stack stTested = new Stack(st);
    Stack stExpected = new Stack();
    Assert.assertTrue(strA. equals(stTested. pop()));
    Assert.assertTrue(stExpected. equals(stTested));
}
```

Listing Four

```
/** Test for method: push(Object)
 * @see Stack#push(Object)
 * @author Jtest
 */
public void testPush2() {
    Object t0 = new Object();
    Object[] var0 = new Object[] {
    };
    Stack THIS = new Stack(var0);
    // jtest_tested_method
    THIS.push(t0);
    boolean var1 = THIS.equals((Stack) null);
    assertEquals(false, var1); // jtest_unverified
}
```

Listing Five

```
[Test]
public void testPopEmpty()
{
    Stack st = new Stack();
    Assert.True(st.pop()==null);
}
```

Listing Six

```
[TestMethod]
public void testPushFull(TestMethodRecord tmr)
{
    String strA = "a";
    Stack s1 = new Stack();
    for(int i=0;i<Stack. STACK_MAX;i++)
    {
        s1.push(strA + Convert. ToString(i));
    }
    Stack s2 = new Stack(s1);
    s1.push(strA);

    tmr. RunTest(s1.equals(s2),"Testing push to a full Stack");
}
```

DDJ

Dissecting Error Messages

Vigorous error handling and good error messages are the key

ALEK DAVIS

Error messages are the most important pieces of information users get when encountering application failures. Good error messages identify the root cause of problems and help users correct them. Bad error messages cause confusion. As an enterprise application developer who frequently participates in software troubleshooting, I see many error messages—and most of them are bad. They are either too generic, ambiguous, misleading, or just plain wrong. Bad error messages cause us to spend long hours investigating problems, which could've been corrected in a matter of minutes had they only been accompanied by meaningful error messages.

In this article, I explain how to build good error messages and present examples showing how to report errors in C/C++ and C#. In the first part of the article, I outline the guidelines covering the most important, but frequently neglected, aspects related to error reporting. The second part describes code samples.

Errors & Error Messages

The right approach to error handling begins with application design. Before writing code, the project team must define a common error-handling methodology, which makes it easier for each developer to process errors. This methodology must include coding standards specifying how errors should be treated (generated, captured, displayed, and stored) and a common library (API) that takes care of the low-level error-handling minutiae. Having a defined standard and available API not only improves developer productivity, but gives us more time to think about the content (meaning) of error messages instead of the implementation details.

Building meaningful error messages is just one aspect of good error handling. Error processing also involves tracing, logging, debugging, and more. Whether you are building Windows GUI applications or web sites, make sure you display error messages in a consistent manner. For Windows GUI apps, use a standard dialog box, which lets users view error details and suggests ways to correct the problem if this information is available.

Displaying error messages in web apps presents more challenges. Unless you have a better approach, show messages at the top of the page or in a JavaScript pop-up dialog. If you include error messages at the top of the web pages, always display them in the same location (relative to page layout) and clearly identify messages as errors. To minimize work, implement the error display area as a common control responsible for rendering error information.

If you choose the JavaScript option, you can follow the example in Listing One. When the server-side code detects an error condition in this example, it calls a shared function, which renders client-side script displaying an alert box containing error information (notice how the code handles quotes and other special JavaScript symbols). You can test this example using the ASPX project (available electronically; see "Resource Center," page 3).

Applications that run without user interfaces (such as server-side applications) normally save error messages in error logs. Examples of error logs can be text files, Windows Event Logs, and databases. Some client-side programs can also use error logs for storing extended error information, such as exception stacks or error traces.

Because a large number of entries in Windows Event Log can make it hard to navigate, server-side programs should use Event Logs only for catastrophic errors and store messages in log files or databases. When storing error information in database tables, make sure that you use space efficiently. For example, when saving an error message returned by a database provider (such as OLE DB) in a 255-character column, store the error description first and details (name of the provider, ANSI code, and so on) last; otherwise, there may not be enough space to hold the description.

> "The right approach to error handling begins with application design"

When writing error messages, think of the users who read it. Error details that make sense to one group of users can totally confuse another. The users reading your error messages include end users, help desk personnel, and application developers. End users may not know much about the application functionality and technology in general. Help desk representatives (and by "help desk" I mean support organizations that do not have access to the source code) usually know more about software than end users, but not as much as the application developers. Application developers know everything (at least, they think they do).

Error messages displayed to end users must be nontechnical and must not contain information that only makes sense to developers (such as exception call stack, error numbers, names of source files, and line numbers). Nor should details of server-side

Alek is a senior application developer at Intel. He can be contacted at alek.davis@intel.com.

(continued from page 34)
errors (say, failure of the application to retrieve data from a database) be passed to end users. For one thing, end users will not be able to correct server-side problems. Additionally, server-side error details can reveal information about the back end, thereby imposing a security risk. On the other hand, error messages displayed to end users must contain enough information to help correct the problems. For example, if an error occurs because users do not have the write permission to a local directory, the name of the directory and required permission must be included in the error message.

Most error messages read by support teams (both help desk and developers) typically come as a result of server-side errors. A server-side error message must include all details that help identify the root cause of the problem. Be careful with code-specific error details. Although application developers like to include exception stack, line numbers, and other code-specific information, these details are generally not helpful to most troubleshooters because they only make sense to programmers who have the source code in front of them. Because code-specific details can make it harder for nondevelopers to understand the meaning of an error message, I don't recommend mixing them with error descriptions. Either save these details in a separate file, or separate them from the human-readable error description.

What Are Good Error Messages?

Good error messages must identify the failed operation, describe the execution context, and specify the reason for failure. If the error reflects a common or anticipated problem, the message can also tell users how to fix it.

Consider an application that creates user accounts for a group of customers, where customer data are pulled from an external system. Account creation is a two-step process—creation of a user profile and a mailbox. Imagine that the application fails to create the mailbox for one user.

To identify the failed operation, an application must name the top-level task it has not been able to complete. Assuming that the problem does not cause failure for the whole batch, the top-level operation is user-account creation. The message must say something like: "Cannot create an account for user X," where X uniquely identifies the user. Identifying the user can help the support team to quickly pull customer data.

After describing the failed operation, the error message must specify the execution path (context) that caused the error. The execution context lists all logical operations that lead to the failure (excluding the topmost operation, which has already been mentioned). You can think of the execution path as the exception stack without code-specific details. In this example, the execution context can be: "Failed to create mailbox for Y," where Y specifies the customer's e-mail address. If the mailbox creation contains substeps, the description of all failed substeps on the execution stack must be appended. The description of each step in the execution context must be unique.

Finally, the reason causing the last step in the execution context to fail must be specified. This information is typically returned by the failed API. For example, it can be retrieved via the *GetLastError* and *FormatMessage*, or obtained from the exception details. The complete error message must read: "Cannot create an account for user X. Failed to create mailbox for Y. Mailbox already exists." Notice that all parts of the error message are written in complete sentences separated by periods, making it easier for the user to follow the steps that caused the failure.

To build a good error message, you need to provide sufficient details, but still avoid duplicate information. Achieving this goal is not easy because errors normally occur deep in the call stack so the type of information that must be returned by error-handling code may not be obvious. The challenge is to return sufficient information from every error handler on the stack.

For example, look at the pseudocode account creation example in Listing Two. If you are writing a function *CreateMailbox*, which message should the error handler of the *CreateMailbox* method pass to the caller? In particular, should it include "Failed to create mailbox for Y" or should this message be generated somewhere else? Keep in mind that a developer responsible for the implementation of the *CreateMailbox* method may not know who will call it.

If you face a similar dilemma, follow this rule: When returning an error message from a function, do not describe the main operation performed by this function. Instead, identify the step on which this function fails and include the error message returned by this step. If function *A* calls function *B*, which calls function *C*, and function *C* causes an error in step *X*, error messages returned by each of these functions should be similar to those in Table 1. Following this rule, the functions in the account creation example would return the error displayed in modified pseudocode in Listing Three.

Although I recommend not including code-specific details, sometimes they may be needed. If a problem is escalated to the development team, a developer may want to know such error details as the exception stack, line number, file name, and so on. To accomplish this, you can make the format of error messages customizable and adjust it at runtime.

Message Returned	Message appended			
	in Main	in A	in B	in C
from C				**Step X failed: Error reason.**
from B			**Cannot do C.**	Step X failed: Error reason.
from A		**Cannot do B.**	Cannot do C.	Step X failed: Error reason.
from main	**Cannot do A.**	Cannot do B.	Cannot do C.	Step X failed: Error reason.

Table 1: *Sample returned error messages.*

BuildMessage*	Creates a formatted message in a dynamically allocated buffer (on the heap).
DeleteMessage*	Deletes memory allocated for a character string on the heap after verifying that it is not NULL.
TryDeleteMessage*	Same as DeleteMessage, except executes within a try…catch block.
GetWin32ErrorMessage	Generates an error description for a specific Windows error code or a current system error.
GetComErrorMessage	Generates an error description for the specified COM error code (HRESULT) and the information retrieved from the COM Error object (IErrorInfo interface).
SetComError	Sets COM error information for the current thread. If there is a pending COM error available via the COM Error object (IErrorInfo interface), this function will also include its description in the new error.
DebugLog	Logs a formatted debug message to a file.

Table 2: *Library functions related to error processing. *ANSI and Unicode versions of these functions are also available.*

Implementing Error Messages In C/C++

Compared to .NET languages, C/C++ does not offer rich error-handling facilities. You

can pass error messages as function parameters or return them as exceptions using structured exception handling (SHE) or C++ exception handling. You can use *GetLastError* in conjunction with *FormatMessage* to get the description of the last failed system call. When making COM calls, you can obtain error information from the COM *Error* object or other COM sources, such as the OLE DB *Error* object.

The hard part about handling errors in C/C++ is memory management. To avoid dealing with memory allocation issues, you can use string classes available in MFC or STL, but this may not always be an optimal approach. In the following example, I show how to build messages in dynamically allocated memory buffers using the basic Win32 APIs (not relying on MFC or STL). I also explain how to process system and COM errors and pass error messages between functions.

The C/C++ sample project (available electronically) builds a static library that implements several methods that can be used to process error messages. To link this library to your project, add it to the project's library settings, making sure that you use the right release target (ANSI or Unicode). In the source code, include a reference to the common.h file, which is located in the Include folder of the project. It defines function prototypes and several helpful macros. Table 2 lists the library's functions related to error processing.

The *BuildMessage* function lets you create formatted messages of arbitrary sizes. It works similar to *sprintf*, but *BuildMessage* writes data to a dynamically allocated memory (on the heap). The function takes three parameters: address of a pointer to a memory buffer, which holds formatted messages; message format; and optional message arguments. There are three versions of *BuildMessage* that work on *TCHAR*, ANSI, and Unicode strings.

BuildMessage dynamically allocates memory if needed. To free memory allocated by *BuildMessage,* you can call the corresponding version of the *DeleteMessage* (or *TryDeleteMessage*) function. This is how *BuildMessage* can be used:

```
TCHAR* pszErrMsg = NULL;
for (int i=0; i<5; i++)
{
    BuildMessage(&pszErrMsg, _T
          ("#%d: Error in file %s...")
                , i, __FILENAME__);
    _putts(pszErrMsg);
}
TryDeleteMessage(&pszErrMsg);
```

When using *BuildMessage*, follow three simple rules:

- Never pass the address of a static character array (stack variable) as the first parameter. *BuildMessage* assumes that the first parameter references a heap variable, so it uses the *_msize* function to check the amount of memory allocated for it and calls *realloc* to allocate additional bytes if the buffer is too small. (If the memory buffer has already been allocated and is sufficient to hold the formatted message, *BuildMessage* reuses it.)
- Before passing a nonallocated memory buffer, always set the value of the pointer (not the address) to NULL; otherwise, the function causes a memory-access violation.
- Always free memory allocated by *BuildMessage* when you no longer need it.

Listing Four is pseudocode illustrating how to use *BuildMessage* to concatenate error information passed between function calls. I find it helpful to follow the convention of always using the first function parameter to pass error messages. If you prefer to pass error information via exceptions, you can still use *BuildMessage* to format message strings, just don't forget to free memory.

Processing Error Information In C/C++

Now that you have a method to easily format error messages, you can generate or retrieve error information via the *GetWin32ErrorMessage*, *GetComErrorMessage*, and *SetComError* functions. The first two methods can be used to retrieve error information from a system or COM call. *SetComError* provides a capability to return COM errors to COM client via the COM *Error* object.

GetWin32ErrorMessage returns the formatted description of the specified Windows error code. If the description is not found, it generates a generic message that includes the provided error number. The function takes one required and two optional parameters. The first parameter is used to hold the generated error message (similar to *BuildMessage*). The second parameter is used to pass the error number. If the error number is not provided or set to zero, *GetWin32ErrorMessage* calls *GetLastError* to retrieve the system error code. The third parameter indicates whether the error number should be included in the error message.

GetComErrorMessage is similar to *GetWin32ErrorMessage*, except in addition to retrieving the description of the HRESULT error value passed as a second (optional) parameter, it also attempts to obtain information from the COM *Error* object (*IErrorInfo* interface). If the HRESULT value indicates success, it is ignored and the function only gets the information from the COM *Error* object. You must always free memory buffer allocated by *GetWin32ErrorMessage* and *GetComErrorMessage*.

ApplicationExceptionInfo	Custom error message formatter for the System.ApplicationException class.
COMExceptionInfo	Custom error message formatter for the System.Runtime.InteropServices.COMException class.
ExceptionFactory	Generates a formatted error message and uses it as a description of an exception created via one of the three overloaded System.Exception constructors.
ExceptionInfo	Custom error message formatter for System.Exception and all exception classes that do not have explicitly defined Custom error message formatters. This class also serves as a base class for other Custom error message formatters.
ExternalExceptionInfo	Custom error message formatter for the System.Runtime.InteropServices.ExternalException class.
FormatFlags	Defines enumeration flags for exception details, which will be included in the error message.
Helper	Implements shared helper methods.
HttpExceptionInfo	Custom error message formatter for the System.Web.HttpException class.
IMessageFormatter	An interface defining the methods responsible for formatting error messages, which must be implemented by Custom error message formatters.
MessageFormatter	Implements helper functions handling message formatting.
OdbcExceptionInfo	Custom error message formatter for the System.Data.Odbc.OdbcException class.
OracleExceptionInfo	Custom error message formatter for the System.Data.OracleClient.OracleException class.
SqlExceptionInfo	Custom error message formatter for the System.Data.SqlClient.SqlException class.
SystemExceptionInfo	Custom error message formatter for the System.SystemException class.
WebExceptionInfo	Custom error message formatter for the System.Net.WebException class.
Win32ExceptionInfo	Custom error message formatter for the System.ComponentModel.Win32Exception class.

Table 3: *Library classes.*

COM methods typically return error information via the COM *Error* object. *SetComError* lets you do it easily and also offers some extras. If you are writing a COM object, which encounters an error when making an internal COM call (such as when calling another COM object), you may want to combine your own error message with the error information retrieved from the failed COM call and pass the result to the client via the COM *Error* object. *SetComError* does just that. It first calls the *GetComErrorMessage* to retrieve error information from the specified HRESULT code and the global COM *Error* object. Then it appends the returned error information—if any—to the message passed via the first parameter and sets this error message as a description field of the *ICreateErrorInfo* interface generated for the current thread. The last two optional parameters can be used to specify the class and interface IDs (CLSID and IID) of your COM object. When using *GetComErrorMessage* or *SetComError*, you must define the *_WIN32_DCOM* precompiler directive in the project settings.

In addition to the three methods just mentioned, you can use the *DebugLog* method to print formatted debug messages to a log file. This function can be handy if you cannot step through the program's source code in Visual Studio IDE. Source code comments describe how *DebugLog* works.

Implementing Error Messages in C#

The .NET Framework offers a much better error-handling methodology that is based on .NET exceptions. Although the concept of exception is not new—you can use exceptions in C/C++—.NET exceptions have a useful feature: They can be nested. This makes it possible to easily pass and retrieve error information. Reflection is another helpful .NET feature, which makes it possible to programmatically retrieve the context of exception at runtime. And of course, using .NET you are free from memory-management hassles.

Enumeration Value	Description
Default	Include exception details identified by the default setting.
Detailed	Include all available exception details.
ErrorCollection	If an exception contains a collection of errors, such as System.Data.SqlClient.SqlException.Errors, include the information about each error in the collection. When this flag is set, all flags specified in the format bitmask will be applied to the error message describing each error in the collection.
Message	Include System.Exception.Message.
Source	Include System.Exception.Source. For System.Data.OleDb.OleDbException, this shows the name of the OLE DB provider; for System.Data.Odbc.OdbcException, it displays the name of the ODBC driver.
Method	Include name of the failed method.
Type	Include name of the exception type (without the namespace), such as "SqlException."
ErrorCode	Include exception-specific error code, such as System.Data.OleDb.OleDbException.ErrorCode, System.Data.OracleClient.OracleException.Code, System.Runtime.InteropServices.ExternalException.ErrorCode, and System.Net.WebException.Status.
Severity	Include error severity, such as System.Data.SqlClient.SqlException.Class.
State	Include database state, such as System.Data.SqlClient.SqlException.State.
MessageNumber	Include message number, such as System.Data.SqlClient.SqlException.Number.
Procedure	Include name of the failed stored procedure, such as System.Data.SqlClient.SqlException.Procedure.
Server	Include name of the server executing the failed call, such as System.Data.SqlClient.SqlException.Server.
NativeError	Include native error code, such as System.Data.OleDb.OleDbError.NativeError or System.Data.Odbc.OdbcError.NativeError.
AnsiCode	Include ANSI code, such as System.Data.OleDb.OleDbError.SQLState or System.Data.Odbc.OdbcError.SQLState.
StatusCode	Include status code, such as System.Web.HttpResponse.StatusCode.
Brief	Same as Message | ErrorCollection.
Details	Same as Source | Method | Type | ErrorCode.
DatabaseDetails	Severity | State | MessageNumber | Procedure | LineNumber | Server | NativeError | AnsiCode.
HttpDetails	Same as StatusCode.

Table 4: *FormatFlags error options.*

The C# sample (available electronically) builds a .NET class library that simplifies error reporting. Using this library, you can:

- Format error messages using error details retrieved from exception objects and inner exceptions.
- Get extended error information from complex exception classes, such as *SqlException*, *WebException*, and others.
- Customize the format of error messages to include or exclude such exception details as source, type, error code, method, and others.
- Extend the library to customize message formatting for your own exception classes.

Table 3 lists some of the library's classes, and Figure 1 is the class diagram. The C# project comes with an HTML help document.

ExceptionInfo is the primary class responsible for retrieving error information from exceptions. This is how you would use *ExceptionInfo* to retrieve error information from a *SqlException* object including all inner exceptions and error details provided in the collection of *SqlError* objects:

```
try
{
 ...
}
catch (SqlException ex)
{
  Console.WriteLine(
    ExceptionInfo.GetMessages(
      ex, (int)FormatFlags.Detailed));
}
```

The *GetMessages* method uses the *FormatFlags* enumerator, which identifies error details you want to retrieve. *FormatFlags* currently supports 14 error detail options and a number of flag combinations (see Table 4). If you want to retrieve more than one error detail, you can combine multiple format flags in a bitmask or use a predefined mask. You do not have to specify format flags on every call, but instead set it only once via the *SetFormat* method.

SetFormat is implemented in the abstract *MessageFormatter* class, which is a parent of *ExceptionInfo*. I used the *MessageFormatter* class to separate message formatting logic from exception processing handled by *ExceptionInfo*. *MessageFormatter* includes a number of methods, which make it easier to build error messages from exception details. *MessageFormatter* implements the *IMessageFormatter* interface.

The *IMessageFormatter* interface defines two *FormatMessage* methods: One uses explicitly specified message format flags, the other uses the default setting. The *FormatMessage* method that uses the default format flags is already implemented in the

FormatMessage class; the other version of *FormatMessage* is an abstract method, which is implemented in *ExceptionInfo* and overridden in *SqlExceptionInfo*, *OdbcExceptionInfo*, and other exception-specific formatter classes.

The library provides two utility classes:

- The *Helper* class is a general-purpose wrapper for frequently used operations.
- *ExceptionFactory* can be used to throw any type of exception with a message that can be built using a message format string and optional message parameters (basically, it combines *String.Format* with throwing an exception).

The *ExceptionInfo* class serves as a custom formatter for the base *Exception* class. In addition, it handles the formatting of error messages for exception classes, which do not have custom formatters. By custom formatters, I mean helper classes such as *SqlExceptionInfo*, which know how to retrieve error details and format error messages for specific types of exceptions, such as *SqlException*.

ExceptionInfo can also be used to retrieve error information from any exception, including the ones that have custom formatters. It dynamically detects whether the exception class has a custom formatter; if so, it uses it to format the error message. If the exception class does not have a custom formatter, *ExceptionInfo* checks its parent and grandparents until it finds the one that has a custom formatter.

For example, say that *ParentException* is derived from *ApplicationException*, *ChildException* is derived from *ParentException*, and *ParentExceptionInfo* implements a custom formatter for *ParentException* (Figure 2). If you call *ExceptionInfo.GetMessage(new ChildException())*, *ExceptionInfo* uses the *FormatMessage* method implemented by *ParentExceptionInfo* to generate the error message.

How does *ExceptionInfo* find the right custom formatter for a given exception type? It uses a naming convention along with the .NET Framework feature, letting code invoke class instances; which types are determined at runtime. The dynamic invocation is accomplished via the *Activator.CreateInstance* method call that takes the name of the class as a parameter. *ExceptionInfo* assumes that the custom formatter is named after the exception class with the postfix "Info," such as *SqlExceptionInfo* for *SqlException*. The custom formatter class must belong to the same namespace as the *ExceptionInfo* class. To see how *ExceptionInfo* invokes the custom formatter, see the *ExceptionInfo.GetFormatter* method.

ExceptionInfo provides a number of helpful methods. In addition to *GetMessage*, which retrieves the error message for the current exception, it implements several overloaded *GetMessages* and *GetMessageArray* methods. Both *GetMessages* and *GetMessageArray* functions can retrieve error information, not only from the immediate exception, but also from inner exceptions. You can specify from which level of inner exceptions you want to start and how many levels the method must process. *GetMessages* returns error messages in a single string (messages are formatted as sentences separated by a single whitespace character with all unnecessary whitespaces removed). *GetMessageArray* returns error messages in a string array. *GetMessages* and *GetMessageArray* are handy if you want to display only parts of error information. For example, you can use them to retrieve the information about inner exceptions in the error details field of the error message box.

Finally, *ExceptionInfo* implements the *GetStackTraces* and *GetStackTraceArray* methods, which work like *GetMessages* and *GetMessageArray*, only these methods retrieve exception stack information from the current and inner exceptions.

Custom Exception Formatters

The library provides a number of custom formatters for exception classes that contain more information than a basic exception. For example, exceptions returned by data providers (such as *SqlException*) can include such details as the name of the failed stored procedure, error severity, ANSI code, and others. A custom formatter knows how to retrieve the requested details from the exception object and display these details in the formatted error message. This is accomplished by overriding the *FormatMessage* method.

To understand how to implement a custom formatter for an exception class not covered by the library, consider one of the custom formatters using *SqlExceptionInfo*.

SqlExceptionInfo is a custom formatter for *SqlException*. It is derived from *ExceptionInfo* and defined under the same namespace. *SqlExceptionInfo* overrides one function—*FormatMessage*. This function receives two parameters: the exception object and the message format flags (bitmask).

To make it easier to access exception-specific members, *FormatMessage* typecasts the exception object from *Exception* to

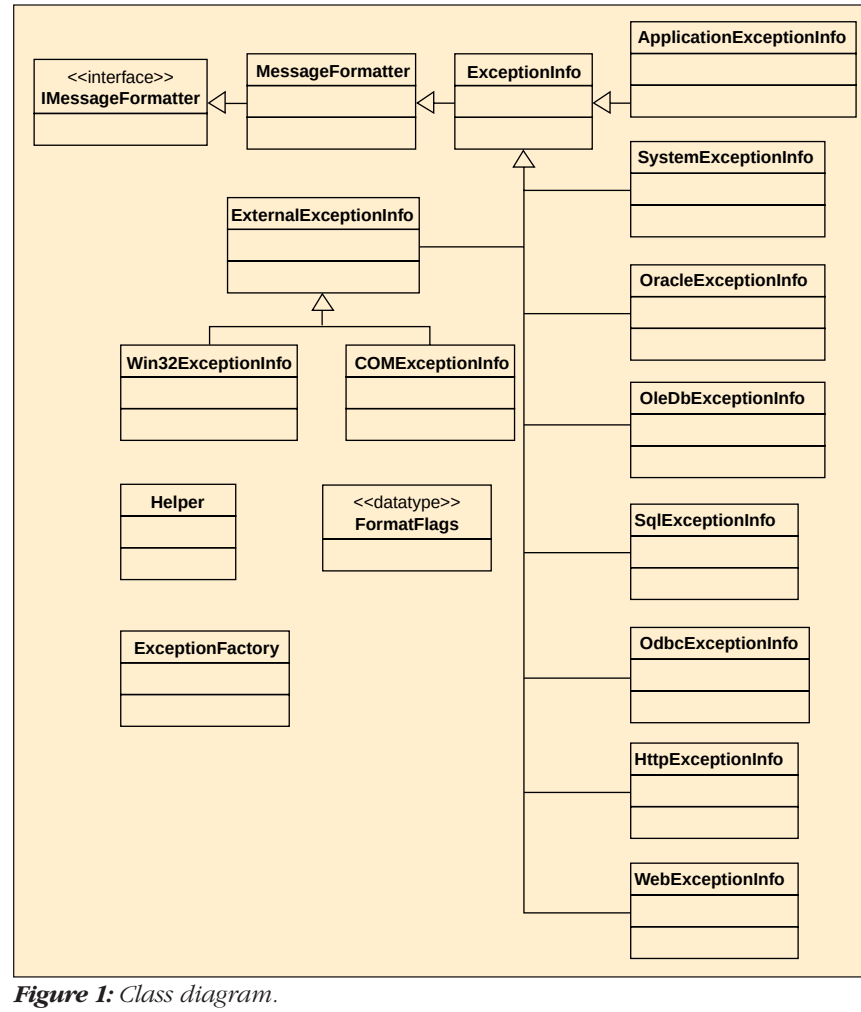


Figure 1: *Class diagram.*

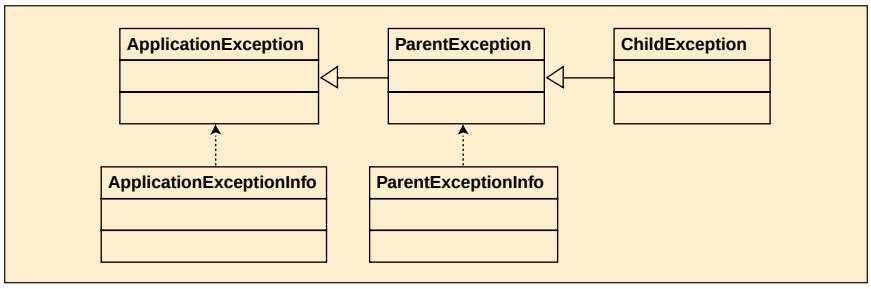


Figure 2: *Custom formatter for* ParentException.

SqlException. Then it verifies the message format flags, and if the value of the bitmask indicates the default (*FormatFlags.Default*), it sets the bitmask to the value of the static member of the *MessageFormatter* class via the *MessageFormatter.VerifyFormat* method. Finally, it processes error information.

Error information available in *SqlException* can be retrieved from the class members, such as *State*, *LineNumber*, *Procedure*, and others, as well as from the collection of *SqlError* objects accessed via the items of the *Errors* member. According to the documentation, the first item in the *SqlError* object collection shares some of the details with the members of the *SqlException* object. For example, the value of *SqlException.Class* member is the same as *SqlException.Errors[0].Class*. To avoid duplicate information, *SqlExceptionInfo* retrieves all information available from the *Errors* collection. After these errors are processed, it adds the details, which are only accessible via the main class members.

Before adding information about a particular exception detail, *SqlExceptionInfo* checks whether a corresponding flag in the specified message format is set; if it is not, the detail will not be included. *SqlExceptionInfo* uses methods inherited from *ExceptionInfo* (and *MessageFormatter*) to build formatted messages for exception details. For additional information, see comments in the source code.

If you implement a custom formatter for an exception class not included in the library, extend the project:

1. Add a class for custom formatter. Name this class by appending "Info" to the name of the corresponding exception class. It must be derived from *IMessageFormatter* (or any other class derived from *IMessageFormatter*, such as *ExceptionInfo*) and created under the same namespace.
2. Override the *FormatMessage* method and implement the logic to retrieve exception details and display them in a formatted error message.

If you need to specify additional fields in the message format bitmask, either extend the *FormatFlags* enumerator or set the unused bits (the format mask is handled as an integer value). I was planning on adding a custom formatter for *SoapException*, but decided not to because *SoapException* is very customizable (SOAP exception details can be passed via XML nodes). If your application makes SOAP calls, I suggest implementing your own version of the custom formatter.

Conclusion

Although you cannot totally eliminate errors from software, vigorous error handling and good error messages can help you reduce the impact of these errors on the users. The better job you do reporting errors, the less time your team will spend troubleshooting them.

DDJ

Listing One

```
//---------------------------------------------------------------------
// $Workfile: BasePage.aspx.cs $
// Description: Implements the BasePage class.
// $Log:  $
//---------------------------------------------------------------------
using System;

namespace ErrorSample
{
/// <summary>
/// Base class implementing common utility functions reused by
/// different pages belonging to this Web application.
/// </summary>
/// <remarks>
/// Page classes on this site must derive from
/// <see cref="ErrorSample.BasePage"/>, not the usual
/// <see cref="System.Web.UI.Page"/>.
/// </remarks>
public class BasePage: System.Web.UI.Page
{
    // We need to keep the counters of the popup script blocks.
    // (Actually, we can distinguish between client-side and
    // start-up scripts, but why bother?)
    private int _errorScriptCount          = 0;
    private string _errorScriptNameFormat = "_errorScript{0}";
    /// <summary>
    /// Formats string and replaces characters, which can break
    /// JavaScript, with their HTML codes.
    /// </summary>
    /// <param name="message">
    /// Message or message format.
    /// </param>
    /// <param name="args">
    /// Optional message parameters.
    /// </param>
    /// <returns>
    /// Formatted string which is JavaScript safe.
    /// </returns>
    private static string FormatJavaScriptMessage
    (
        string          message,
        params object[] args
    )
    {
        // Make sure we have a valid error message.
        if (message == null)
            return String.Empty;
        // If we have message parameters, build a formatted string.
        if (args != null && args.Length > 0)
            message = String.Format(message, args).Trim();
        else
            message = message.Trim();
        // Make sure we have a valid error message.
        if (message.Length == 0)
            return String.Empty;
        // Back slashes, quotes (both single and double),
        // carriage returns, line feeds, and tabs must be escaped.
        return message.Replace(
            "\\", "\\\\").Replace(
                "'", "\\'").Replace(
                    "\"", "\\\"").Replace(
                        "\r", "\\r").Replace(
                            "\n", "\\n").Replace(
                                "\t", "\\t");
    }
    /// <summary>
    /// Shows a formatted error message in a client-side (JavaScript)
    /// popup dialog.
    /// </summary>
    /// <param name="message">
    /// Error message or message format.
    /// </param>
    /// <param name="args">
    /// Optional message arguments.
    /// </param>
    /// <remarks>
    /// Error popup will be rendered as the first element of the
    /// page (form).
    /// </remarks>
    public void ShowErrorPopup
    (
        string          message,
        params object[] args
    )
    {
        ShowErrorPopup(true, message, args);
    }
    /// <summary>
    /// Shows a formatted error message in a client-side (JavaScript)
    /// popup dialog.
    /// </summary>
    /// <param name="showFirst">
    /// Flag indicating whether the error message must be rendered
    /// as the first element of the page (form).
    /// </param>
    /// <param name="message">
    /// Error message or message format.
    /// </param>
    /// <param name="args">
    /// Optional message arguments.
```

```
    /// </param>
    public void ShowErrorPopup
    (
        bool             showFirst,
        string           message,
        params object[] args
    )
    {
        // Build message string which is safe to display in JavaScript code.
        message = FormatJavaScriptMessage(message, args);
        // If we did not get any message, we should not generate any output.
        if (message.Length == 0)
            return;
        // Generate a unique name of the start-up script.
        string scriptBlockName = String.Format(
        // Generate HTML for the script.
        string scriptHtml = String.Format(
                            "{0}" + "<SCRIPT Language=\"JavaScript\">{0}" +
                            "<!--{0}" + "alert(\"{1}\");{0}" + "-->{0}" +
                            "</SCRIPT>{0}", Environment.NewLine, message);
        // Generate script opening a popup with error message.
        if (showFirst)
            RegisterStartupScript(scriptBlockName, scriptHtml);
        else
            RegisterClientScriptBlock(scriptBlockName, scriptHtml);
    }
}
}
//---------------------------------------------------------------------
// $Workfile: Default.aspx.cs $
// Description: Implements the DefaultPage class.
// $Log:  $
//---------------------------------------------------------------------
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ErrorSample
{
/// <summary>
/// Implements the default Web page.
/// </summary>
public class DefaultPage: BasePage
{
    protected System.Web.UI.WebControls.Button btnTest;
    private void Page_Load(object sender, System.EventArgs e)
    {
    }
    #region Web Form Designer generated code
    override protected void OnInit(EventArgs e)
    {
        // CODEGEN: This call is required by the ASP.NET Web Form Designer.
        InitializeComponent();
        base.OnInit(e);
    }
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.btnTest.Click += new System.EventHandler(this.btnTest_Click);
        this.ID = "Form";
        this.Load += new System.EventHandler(this.Page_Load);
    }
    #endregion
    // Displays error messages when button is clicked.
    private void btnTest_Click(object sender, System.EventArgs e)
    {
        for (int i=0; i<4; i++)
        {
            // Display errors with odd IDs as the first elements of
            // the form and the rest as the last elements of the form.
            bool showFirst = (i & 0x1) == 1;
            ShowErrorPopup( showFirst, "Error '{0}' occurred at:{1}{1}{2}",
                            i, Environment.NewLine, DateTime.Now);
        }
    }
}
}
```

Listing Two

```
bool CreateAccounts()
{
  UserListInfo = GetUserListInfo();
  foreach(UserInfo in UserListInfo)
  {
    if (not CreateAccount(UserInfo))
    {
      Report error.
    }
  }
}
bool CreateAccount(UserInfo)
{
  if (not CreateProfile(UserInfo))
  {
    Report error.
    return false;
  }
  if (not CreateMailbox(UserInfo))
  {
    Report error.
    return false;
  }
  return true;
}
bool CreateProfile(UserInfo)
{
  Create profile.
}
bool CreateMailbox(UserInfo)
{
  Create mailbox.
}
```

Listing Three

```
bool CreateAccounts()
{
  UserListInfo = GetUserListInfo();
  string msg, errMsg;
  foreach(UserInfo in UserListInfo)
  {
    if (not CreateAccount(msg, UserInfo))
    {
      errMsg = "Cannot create account for user "+UserInfo.UserID + ". " + msg;
      ReportError(errMsg);
    }
  }
}
bool CreateAccount(errMsg, UserInfo)
{
  string msg;
  if (not CreateProfile(msg, UserInfo))
  {
    errMsg = "Cannot create profile for " +
                UserInfo.ProfileName +
                ". " + msg;
    return false;
  }
  if (not CreateMailbox(msg, UserInfo))
  {
    errMsg = "Cannot create mailbox for " + UserInfo.MailboxName + ". " + msg;
    return false;
  }
  ...
  return true;
}
bool CreateProfile(errMsg, UserInfo)
{
  if (profile exists)
  {
    errMsg = "Profile already exists.";
    return false;
  }
  Create profile.
  ...
  return true;
}
bool CreateMailbox(errMsg, UserInfo)
{
  if (mailbox exists)
  {
    errMsg = "Mailbox already exists.";
    return false;
  }
  Create malbox.
  ...
  return true;
}
```

Listing Four

```
bool DoThis(TCHAR** ptszErrMsg, ...)
{
  TCHAR* ptszMsg = NULL;
  if (!DoThat(&ptszMsg, ...))
  {
    BuildMessage(ptszErrMsg, _T("Cannot do that. %s"), ptszMsg);
    TryDeleteMessage(&ptszMsg);
    return false;
  }
  return true;
}
bool DoThat(TCHAR** ptszErrMsg, ...)
{
  TCHAR* ptszMsg = NULL;
  if (!DoTheOther(&ptszMsg, ...))
  {
    BuildMessage(ptszErrMsg, _T("Cannot do the other. %s"), ptszMsg);
    TryDeleteMessage(&ptszMsg);
    return false;
  }
  return true;
}
main()
{
  TCHAR* ptszMsg    = NULL;
  TCHAR* ptszErrMsg = NULL;

  if (!DoThis(&ptszMsg, ...))
  {
    BuildMessage(&ptszErrMsg, "Cannot do this. %s", ptszMsg);
    puts(ptszErrMsg);
    TryDeleteMessage(&ptszMsg);
    TryDeleteMessage(&ptszErrMsg);
  }
}
```

DDJ

Debugging Production Software

The PSD library provides a few powerful tools

JOHN DIBLING

It's crucial that the software we write be as close to bulletproof as possible. But production environments are hostile, and it sometimes seems like they were designed to chew up software and spit out a smoking, mutilated mass of worthless bytes. Users often do things we do not expect them to — or worse, told them not to. When our software doesn't do something it was never designed to do — and do it perfectly — users cancel units, never to be seen or heard from again.

Nobody ever said programming was easy. Just for the record, I'll say it now — programming is hard. No matter how good you are, how much experience you have, how many books you read and classes you take, you can't escape inevitability. You will write bugs. I do it every day. I often tell my coworkers that if they aren't writing at least two new bugs each day, they probably aren't working hard enough. When I was a rookie C++ programmer, I thought that the key to writing code that was defect free was to know more C++, more techniques, more tricks. I don't think this anymore, and I'm much happier.

Debugging sessions are fine for detecting the major design flaws and little syntactic errors that crop up during development — buffer overruns, sending the wrong kind of message to some server somewhere, and the like. But what happens when bugs are detected by users in production software? Usually, angry users or administrators phone the help desk, but with little information that helps you debug the problem. You typically know what the symptoms were, because these are what told users there was a bug in the first place. But users are usually unreliable sources of objective or accurate information, and they generally cannot tell you what was happening before the bug occurred. Of course, this is the information you really need. So unless you are lucky and just happen to stumble across an obviously stupid piece of code such as this:

```
QUOTE* pQuote = new QUOTE;
delete pQuote;
pQuote->SetPrice(1.234f);
```

you will probably spend days looking for the bug. Once you get close enough to reproduce it, fixing the defect is usually comparatively simple, and often limited to around one line of code.

The problem is a lack of information. A bug is, by definition, an unknown quantity. Most language-level features that are designed to help diagnose problems are not intended for use in production software. Things like *assert()* become useless or worse in release builds. When you get a call about a bug in production software, it takes forever to identify and fix the problem. Most of the time you spend in just trying to reproduce the problem. If you could identify the state of the universe more quickly, the effort needed to resolve bugs would go down a lot.

The Production Software Debug (PSD) library I present here is a library of utilities designed to identify and diagnose bugs in production software. There are only three main features in the library, but they pack a wallop. Used liberally in production code, the PSD library (available electronically; see "Resource Center" page 3) has helped to significantly reduce the amount of time it takes to fix bugs. Its three main features are:

- *verify()*, a better *assert()*.
- *static_verify()*, a compile-time version of *verify()*.
- *OutputMessage()*, a generic logging mechanism that is easy to use and extend.

verify(): A Better *assert()*

There are few C++ language-level features to help identify bugs, and what precious few do exist are not suitable for production software. One language feature that was added early in the language's evolution was the *assert()* macro. The idea was simple. When a function is executed, you expect the software to be in a sane state. Pointers point to the right thing. Sockets are open. The planets are aligned. *assert()* makes it possible to check these things at runtime easily, to add precondition and postcondition checks to blocks of code.

> "A bug is, by definition, an unknown quantity"

But *assert()* is contrived. If the expression sent to *assert()* is false, it kills your program. Back in the '70s, when software was written by the people who ran it, maybe this kind of behavior was okay. But today, if a wild pointer results in the application going *poof* — well, that's just not going to do at all. Pointers shouldn't be wild in the first place, but the main point is that no matter how much code you write to keep your pointers from being wild, it's not going to be enough. Sometimes they will go wild, anyway. You must come to terms with this fact. It turns

John is a programmer from Chicago, Illinois. He can be reached at jdibling@computer.org.

(continued from page 42)
out that *assert()* isn't really useful at all for dealing with wild pointers in code that was written and tested. It's only useful in testing code that's still in development.

There are three major problems with *assert* that make it unsuitable for production code. The first one I already mentioned—it rips the bones from the back of your running program if a check fails. Second, it has no return value so that you can handle a failed check. Third, it makes no attempt to report the fact that an error occurred. The PSD library's runtime testing utilities address these problems. They are template functions that accept any parameter type that is compatible with *operator!*, and return *bools—true* for a successful check, *false* for a failed check. If the check fails, the test utilities simply return *false* and do not terminate the program or do anything similarly brutal. Like *assert(),* in a debug build, a failed *verify()* will also break into the debugger. But the most significant features of the *verify()* utilities are the tracing mechanisms.

The PSD library includes tracing utilities, and *verify()* uses these tracing utilities to dump a rich diagnostic message when a check fails. The message is automatically generated and output to a place where you can get it.

The diagnostic message is rich, meaning it includes a lot of detailed information. Actually, there isn't much information to include, but all of it is included. Specifically, the message says the exact location of the failed check, including source filename and line number, and the expression that failed, including the actual value of the expression. For example, suppose your program is logging a user into a server, and you have a runtime check to assert that the login succeeded:

```
if( !verify( LOGIN_OK ==
        pServer->Login(jdibling,password))
{
   // handle a failed login attempt here
}
```

If the login did not succeed, this diagnostic message is generated and logged:

```
*WARNING* DEBUG ASSERTION FAILED:
        Application:
  'MyApp.EXE', File: 'login.cpp', Line: 120,
        Failed Expression:
      'LOGIN_OK ==
        pServer->Login("jdibling","password")'
```

This diagnostic message is sent to whatever destinations you configure (one or more), and you can configure whatever destinations you like, including your own proprietary logging utility. By default, the PSD library sends all such messages to three places: *std::cerr*, the MSVC 6.0 debugging window (which is visible in production runs using the DbgView.EXE utility, a freely available utility at http://www.sysinternals.com/), and a cleartext log file. (The name and location of this file is configurable, but by default it is named "log.txt" and saved in the current working directory.) The diagnostic message is extremely helpful in diagnosing problems that occur in customer's machines. It is generally a much simpler matter to acquire a log file from a customer than to try and reproduce the error condition. In addition, it frequently is not enough to know just the failed expression and the location of the failed source code. Usually, you need to know how the universe got in such a state, and the previous output messages that occur when the PSD library is used liberally is of extraordinary significance. For example, I usually want to know what exact version of my software the error occurred in, and I output that information to a log file using the tracing utilities in the PSD library. These two pieces of information taken together are often enough to know just what happened and why.

In the aforementioned code, *verify* is actually a preprocessor macro for the template function *Verify* (note the change in case). Generally, I don't like macros, but in this case, the benefits outweighed any detraction. You could call the *Verify()* template function directly, as it is included in the library interface, but there is little point and I have never seen a reason to do so. Also, if you call *verify()* (the macro version) and the check fails, the diagnostic message includes the filename and line number of the failed check. This is accomplished through macro black magic. If you call *Verify()* (the low-level template function) directly, you lose this benefit and are on your own in trying to figure out which *Verify()* check failed.

There are several other flavors of *verify()* as well, good for common special cases and taking more control over its behavior. One flavor is *noimpl(),* a default handling placeholder for the black holes in your code. The most common example of this is the default handler in a switch statement. In the case where your intent in a switch is to handle every possibility, you often have a default handler to do some default handling when things go wrong. Adding a *noimpl()* call to these blocks triggers a call to *verify(false).* Many otherwise very-hard-to-detect bugs are simply flagged using this feature.

Another flavor of *verify()* is *testme(),* which is kind of like a bookmark. When writing new blocks of code that you intend to test by stepping through manually, just add a call to *testme()* at the beginning of the block. I have found that when I'm writing code that I plan to step

(continued from page 44)
through, it is usually in lots of different places and I tend to lose track of them. *testme()* breaks into the debugger when it is run (just like *verify*) and reminds you where to test.

static_verify()

static_verify() is a version of *verify()* that is "run" at compile time, rather than runtime. The motivation for this device has existed for many years, but the design for it was derived from one presented in Andrei Alexandrescu's book *Modern C++ Design*.

static_verify() is especially useful at detecting when some critical implementation details have changed without your realizing it. Relying on the implementation details of some data structure or object is almost always a bad idea. But in the real world, it happens all the time. Older code, newer programmers, and plain bad designs are everywhere, and our job is to get all of this code to work first, and pontificate about how it isn't pristine later.

This code is guaranteed to work so long as the two user *id* fields are the same size:

```
struct USER
{
   char m_cUID [10];
   char m_cPwd[10];
};
struct LOGIN_MESSAGE
{
   char m_cUID[10];
   char m_cPwd[10];
};
:          :
static_verify( sizeof(USER::m_cUID) ==
      sizeof(LOGIN_MESSAGE::m_cUID) ));
memcpy(user.m_cUID,login.m_cUID,
      sizeof(user.m_cUID));
```

Because it is doing a *memcpy(),* it's going to be fast. It does not matter what the size actually is, and it does not matter what the format of the *char* buffers are (for example, whether they are null terminated, space padded, or whatever). But if one of the *char* buffers is changed in size, the *static_verify()* halts the compiler with an error message, and you can adjust your algorithm to work with the new disparity.

OutputMessage()

OutputMessage() and the other tracing utilities make it easy to generate messages that are sent wherever you want. Use *OutputMessage()* liberally to log the values of variables and parameters, trace the execution path of a function, and so on. Again, the runtime testing utilities also generate calls to *OutputMessage()*.

OutputMessage() works like *sprintf,* so it is easy to use, and chances are pretty good that you already know how to use it. There are flavors of *OutputMessage()* that take additional parameters specifying the destination of the message, options flags, and so on. But the general-purpose *OutputMessage()* takes just a format string and a variable parameter list, just like *sprintf()*.

OutputMessage() can send messages to wherever you want, and it is easy to get it to send a message to somewhere new. Simply define a callback function and register it with the PSD library as such, set a global PSD library option to always send messages there, and you're done. From then on, every time *OutputMessage()* is called, messages will be sent to your routine. You can define numerous destinations and have messages sent to all, one, or none of them. You can also call *OutputMessageEx()* to send a specific message to a specific location.

Conclusion

The PSD library was written in C++ using only standard-compliant features in its interface. It was originally intended for use on Windows platforms and the Microsoft Visual C 6.0 compiler, but there are no platform-specific features in the interface. The implementation of these features in many cases does make use of Windows-specific functions and primitives, as you might expect. But on the whole, it should be easily adaptable to other platforms and compilers.

In production code, where the PSD library was used extensively, the time needed to diagnose, debug, and fix bugs was reduced drastically. There are two keys to reducing debug time:

- Using *verify* to detect errors at runtime. It is possible to simply do a global search and replace in code that currently uses *assert*, changing all instances of *assert* to *verify*. Adding additional calls to *verify* also helps. The normal case of execution for a *verify* is for testing a true expression, and this common case is executed fast. If the expression is true, *verify* consists of one function call, an *if* statement, and a *return* statement. Because of this, *verify* is appropriate to use in time-critical code.
- Logging the state of the running program before problems occur. To debug faulty code, in addition to knowing the failed expression, it is important to know the version of the software, the values of internal variables and function parameters, whether pointers are valid, and so on. Using *OutputMessage()* adds this information to the log and helps reduce debug time.

DDJ

System Verification With SCV

Logic is logic, and testing is testing

GEORGE F. FRAZIER

Testing and debugging embedded software systems and hardware designs provides challenges similar to software test and debug. Logic is logic, whether it is implemented in software or silicon. However, although this is certainly not universally true, the stakes can be higher for hardware systems because of several factors. Market forces are driving much of the electronics industry into the profitable consumer arena. Consumer products need to be easy to use and cheap, and time-to-market is as critical today as ever. Finally, chip designs need to be innovative and differentiated from competitors. These goals, which are often at odds with each other, are running into another technological force—the ever-shrinking transistor. Despite rumors of its imminent demise, Moore's Law continues with frenzy and ever smaller geometries: Gate dimensions have marched down the "submicron" path from 0.65 microns, to 0.5 microns, to 0.25 microns, to 0.18 microns, and now 0.13 microns. The process continues. With each step, the factories that print chips have to retool, and that cost is passed along to chip designers. To provide reasonable cost and function, new chip designs are commonly entire "Systems-on-a-Chip," with previously uncoupled subsystems residing on a single chip that might total over 15 million gates. The cost of failure is high. If a chip comes back faulty from the fabricating plant, the average cost for a respin is approaching $1 million. Recent market research indicates more than half of all new chips require at least one respin, and an even higher percentage of final products have major functional flaws.

The Electronic Design Automation (EDA) industry provides tools that hardware and embedded systems designers use to produce new designs. EDA tools have evolved along with customer demands. On the design side, the last 15 years have been marked by the dominance of hardware-design languages—principally Verilog and VHDL, which support modeling at the RTL-level of abstraction. The Register Transfer Level (RTL-level) tracks system changes at the clock cycle. This is turning out to be too computationally intense for the design and verification of Systems-on-a-Chip that contain over 10 million gates. A new level of abstraction, the Transaction Level (or more generally, the Electronic System Level, ESL), is emerging where simulation is viewed as a behavioral flow of data through a system. Granularity shifts from clock bursts to flow events such as data bursts across a bus. The single transfer of an MPEG from memory to a speaker can be modeled in one discrete event with an associated cost, rather than being broken down into the individual clock bursts and bit transfers that make up the physical event. This provides faster simulation and a quicker turnaround for design decisions.

This new ESL is not only important at the design level but is increasingly more critical at the verification stage that occurs before a chip design is masked into silicon. Historically, it was possible to exhaustively test a circuit model by trying all possible test vectors against a known matrix of results. This black-box testing has parallels in software testing and the same issues arise there—time. With large designs, an exhaustive approach is impossible. Techniques familiar to software testers are being modified and translated to the hardware and embedded software arenas with specific emphasis being placed on the particular problems of massive design spaces. Several ESL technologies have arisen to implement these techniques. I address one of them in this article—SystemC Verification (SCV).

> "SystemC was developed to standardize an ad hoc collection of C-based ESL technologies"

George works on ESL technologies and SystemC at Cadence Design Systems Inc. He can be reached at georgefrazier@yahoo.com.

The SCV library is an open-source class library that works in conjunction with the open-source library SystemC. Both libraries are built on Standard C++. SystemC and SCV are governed by the Open SystemC Initiative (http://www.systemc.org/). SCV is being widely used to verify not only SystemC designs but designs written in Verilog and VHDL, or a combination of all three. Note the similarity to software testing. Chip designs are written in programming languages that are expressive of hardware constructs, the design is verified at a certain level of abstraction, and then other programs translate (similar to "compile") or synthesize the higher level programmatic representations of System function down to a gate-level design that can be fabricated into a chip.

Verification at the ESL

SystemC was developed to standardize an ad hoc collection of C-based ESL technologies so that EDA vendors and IP designers could exchange models and standardize interfaces for better interoperability between products and product flows. This is important for system design, but especially important for system verification, where the goal is to build a reusable infrastructure of test stimuli and models. For a more detailed discussion of SystemC, see my article "SystemC: Hardware Constructs in C++" (*C/C++ Users Journal*, January 2005).

As systems become increasingly more complex, it is no longer possible to exhaustively test designs from a black box perspective. Modern verification techniques are designed to be integrated with the development process and implemented by experts who know most about what should and should not be happening at any stage of the design. SystemC itself lacks some of these methods. However, C++ in general is a convenient language for generating test benches, even if the IP models are written in Verilog or VHDL. With SystemC, ESL verification can be done across the entire lifecycle of a project, and verification blocks can be reused between projects and for both block verification and examination of design tradeoffs. These facts led to the development of a separate set of libraries on top of SystemC for Verification. Figure 1 illustrates a transaction-based verification scheme based on SCV. Here, Tests communicate via a Transactor with the model of the system design. Tasks are Transaction-level events; that is, above the RTL-level. If the design is at the RTL-level because, for example, it is written in Verilog or VHDL, the signals between the Transactor and the model are RTL-level signals. A Transactor can be thought of as an adaptor that translates communications between the various levels of abstraction. For example, a Transactor can translate a high-level operation, such as a function call or bus transfer into its component signals or data bursts that are clock accurate.

Randomization

Because it is not possible to test all combinations of possible inputs (or stimuli) of a design, a subset of stimuli is chosen for system verification. One approach to this is to construct tests by hand. This is helpful for bug tracking (a unit test is introduced to ensure that subsequent changes don't "unfix" the improvement), but for thorough coverage, hand-constructed tests can be both limiting and biased. Randomization lets test vectors be picked randomly with values ascribed possibly based on certain bounding criteria.

Unconstrained randomization is unbounded: Data values have an equal probability of occurring anywhere in the legal space of the data type. This is similar to using C's *rand()* function to generate a random integer value between 0 and RAND_MAX.

Weighted randomization weights the probability so that the distribution of data values is not uniform. This is tunable. An example would be setting a 75 percent probability that an integer input will take on a value between 0 and 50 and a 25 percent probability that the input will be between 51 and 100.

A more sophisticated randomization is constraint-based randomization where an input is constrained to a range of its legal values by a set of rules specified by constraint expressions. The constraint expressions can be simple ranges or a complex expression that may include other variables under constraint and subexpressions, and so on.

The first class of importance in SCV randomization is the *scv_smart_ptr<T>* class, a container that provides multithreaded memory management and access and automatic garbage collection for both built-in C++ types and SystemC types, such as *sc_int*, *sc_uint*, and the like. Smart pointers are essential in SCV because SystemC is a multithreaded library where memory might be accessed by more than one thread or even more than one process. The *get_instance()* method of *scv_smart_ptr<T>* provides direct access to the underlying data object (this should be used with caution). Because it is problematic to apply the core algorithms of randomization to data structures that can dynamically change in size, SCV randomization is limited to data types with a fixed size (this excludes lists, trees, and so on). However, users can create smart pointers for all extended types, including user-defined *struct*s. Finally SCV provides many general-purpose methods for manipulating smart pointers such as copying a smart pointer (include "deep" copy).

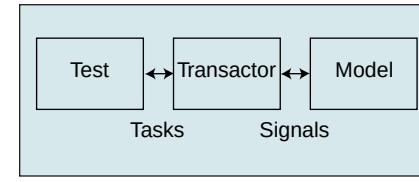


Figure 1: *A transaction-based verification scheme based on SCV.*

Listing One is an example of using weighted randomization in SCV. A system is simulated where operations exist to RECEIVE, STORE, DECODE, MANIPULATE, ENCODE, and SEND a *jpeg* object. The operations are named in an *enum* that is used as a smart pointer. Weighted randomization is implemented in SCV with the class *scv_bag,* which creates distributions, and the smart pointer method *set_mode(),* which assigns a distribution to an input. I assign weights to each operation with the *add()* method of *scv_bag* (for convenience, a total of 100 is used, so it is obvious that, for example, there is a 40 percent probability in any event cycle of variable *jpg* taking on the value of MANIPULATE). For clarity, the example omits the extensions definition that would be required to do randomization on the user's *enum*. Weighted randomization is most useful when an input has a known set of legal values and it is desired to favor certain ranges differently than others.

The richness of randomization in SCV is achieved through the constraint classes, which are containers for randomization objects that have complex constraints. Constructing a class that contains both a data type and the constraints on that type separates the two and allows the use of C++ class inheritance. While a detailed treatment of constraint-based randomization in SCV and the mechanisms of an efficient constraint solver is beyond the scope of this article, I can share one small example.

Constraint classes derive from *scv_constraint_base* and must contain at least one *scv_smart_ptr* (each such smart pointer must be instantiated on a simple object, no nested smart pointers or hierarchy is allowed). Listing Two shows the creation and use of a constraint class. Here, a simple bounded integer is represented by three constraints. Note that the use of the *scv_constraint_base::next()* generates instances of the constrained object. This call randomizes all smart pointers in the constraint class.

Transaction Monitoring and Recording

One of the basic debugging and analysis functions of verification is to record

and report the results of operations within the Transactor. This is done via transaction monitoring and recording with the SCV Transaction API. The output in SCV is text based. In SCV, you control what happens during transaction recording by registering callbacks. For example, to do text recording, users call *scv_tr_text_init()*, which registers the appropriate callbacks for text recording. Similar strategies can be used to change how transactions are recorded. For instance, to record to an SST2 database (a vendor-specific signal database provided by Cadence Design Systems) you call *cve_tr_sdi_init()* to register those callbacks. Text recording can be slow, so this is a powerful way to extend transaction recording. Monitoring can be done dynamically or the output can be dumped out for postprocessing.

Transactions are events that have a beginning and ending time and an associated data set. Important classes in SCV transaction recording are:

- *scv_tr_generator*, a class for generating transactions of a specific type.
- *scv_tr_stream*, a grouping of related or overlapping transactions.
- *scv_tr_db*, a transaction database that collects a group of transaction streams.
- *scv_tr_handle*, a handle to a transaction.

Data Introspection

An important facilitator for randomization, constrained randomization, and transaction recording is the ability to perform certain operations on complex C++ and SystemC data types. The C function *rand()* generates a random integer. SCV allows similar randomization of higher order C++, SystemC, and SCV classes and types with a form of data introspection. Introspection lets a program gain knowledge about the properties of an initially unknown data object. Thus, in the example of *rand()*, this can be extended to complex types by using introspection to determine the name and data types of the data members of an object, and applying type-appropriate customizations of *rand()* on those composite members (often wrapped in an *scv_smart_ptr*). In SCV, data introspection is implemented with template specialization so code can work with a data object without explicit type information at compile time. Without this powerful SCV feature, users would need to implement this with custom code for every class in the verification design.

The data introspection facility provides a standard abstract interface, *scv_extensions_if*, from which a data object can be analyzed and manipulated. The *scv_extensions* template extends data objects so that they support the abstract interface via partial template specialization. SystemC, C++, and C built-in types have corresponding data introspection types, including:

- Class *scv_extensions<bool>* for *bool*.
- Class *scv_extensions<char>* for *char*.
- Class *scv_extensions<sc_string>* for *sc_string*.
- Class *scv_extensions<T*>* for *pointer*.

scv_extensions has a host of member functions for introspecting the data type. Listing Three is an example where a user-defined type is queried for the number of fields. This is just a small primer on data introspection. Besides static analysis, such as in the aforementioned example, a rich implementation of dynamic analysis is also available in SCV.

Conclusion

SCV is widely gaining support from the ESL community and has already been used to verify chips that have made it to tape out. It is not without competitors in the ESL verification space, namely SystemVerilog and the "e" language. However, as SystemC continues to gain prominence, SCV and the libraries that simplify its use should gain traction and help spur the adoption of Electronic System Level, "preimplementation" Verification in hardware and embedded system design.

DDJ

Listing One

```
enum PROCESS_JPEG_EVENTS {RECEIVE, STORE, DECODE, MANIPULATE, ENCODE, SEND};

int jpeg_stream()
{
   scv_smart_ptr<PROCESS_JPEG_EVENTS> jpg;
   scv_bag<PROCESS_JPEG_EVENTS>       jpg_dist;
   jpg_dist.add(RECEIVE, 10);
   jpg_dist.add(STORE, 10);
   jpg_dist.add(DECODE, 20);
   jpg_dist.add(MANIPULATE, 40);
   jpg_dist.add(ENCODE, 10);
   jpg_dist.add(SEND, 10);

   jpg->set_mode(jpg_dist);

   while (1)
   {
       jpg->next();
       switch (jpg->read())
       {
           case RECEIVE:  jpg_receive(); break;
           case STORE: jpg_store(); break;
           case DECODE: jpg_decode(); break;
           case MANIPULATE: jpg_manipulate(); break;
           case ENCODE: jpg_encode(); break;
           case SEND: jpg_send(); break;
           default: return 1;
       }
   }
   return 1; // never return
}
```

Listing Two

```
// An SCV Constraint Base Class with 3 constraints

class boundary_constraint_class : public scv_constraint_base
{
public:
    scv_smart_ptr<sc_uint> burst;
    scv_smart_ptr<uint> lower;
    scv_smart_ptr<uint> upper;

    SCV_CONSTRAINT_CTOR(boundary_constraint_class)
    {
        SCV_CONSTRAINT (lower() > 100);
        SCV_CONSTRAINT (upper() < 500);
        SCV_CONSTRAINT (burst() >= lower()
                        && burst() <= upper() );
    }
};
// using the boundary class
int use_boundary()
{
    boundary_constraint_class bc ("boundary_constraint_instance");
    // The argument is a name string required by SystemC
    for(int i=0; i < DESIRED_NUMBER_OF_CONSTRAINED_RANDOM_VALUES; ++i)
    {
        bc.next(); //generate values
        cout << "Value of burst: " << bc.burst() << endl;
    }
    return 0;
}
```

Listing Three

```
template <typename T> void fields(const T&obj)
{
    scv_extensions<T> ext = scv_extensions(obj);
    cout << "Our object has " << ext.get_num_fields() << " fields." << endl;
};
class aClass
{
public:
    long I;
    int t;
    char *p;
};
SCV_EXTENSIONS(aClass) {
public:
  scv_extensions<long> I;
  scv_extensions<int> t;
  scv_extensions<char*> p;
  SCV_EXTENSIONS_CTOR(aClass) {
    SCV_FIELD(I);
    SCV_FIELD(t);
    SCV_FIELD(p);
  }
};

void example()
{
  aClass ac;
  fields(ac);
}
```

DDJ

Portability & Data Management

Simplify the reuse of data-management code in new environments

ANDREI GORINE

Whether an embedded-systems database is developed for a specific application or as a commercial product, portability matters. Most embedded data-management code is still homegrown, and when external forces drive an operating system or hardware change, data-management code portability saves significant development time. This is especially important because the lifespan of hardware is increasingly shorter than that of firmware. For database vendors, compatibility with the dozens of hardware designs, operating systems, and compilers used in embedded systems provides a major marketing advantage.

For real-time embedded systems, database code portability means more than the ability to compile and execute on different platforms: Portability strategies also tie into performance. Software developed for a specific OS, hardware platform, and compiler often performs poorly when moved to a new environment, and optimizations to remedy this are very time consuming. Truly portable embedded systems data-management code carries its optimization with it, requiring the absolute minimum adaptation to deliver the best performance in new environments.

Andrei is principal architect at McObject. He can be reached at gor@mcobject.com.

Using Standard C

Writing portable code traditionally begins with a commitment to use only ANSI C. But this is easier said than done. Even code written with the purest ANSI C intentions frequently makes assumptions about the target hardware and operating environment. In addition, programmers often tend to use available compiler extensions. Many of the extensions — prototypes, stronger type checking, and so on — enhance portability, but others may add to platform dependencies.

Platform assumptions are often considered necessary for performance reasons. Embedded code is intended to run optimally on targets ranging from the low-end 8051 family, to 32-bit DSP processors, to high-end Pentium-based SMP machines. Therefore, after the software has been successfully built for the specific target, it is customary to have a performance tuning stage that concentrates on bringing out the best of the ported software on the particular platform. This process can be as straightforward as using compiler-specific flags and optimizations, but often becomes complex and time-consuming and involves patching the code with hardware-specific assembler. Even with C language patches, hardware-optimized code is often obscure and, more importantly, performs poorly on different machines.

Programmers also attempt to maintain portability through conditional code (*#ifdef/#else*) in a master version that is preprocessed to create platform-specific versions. Yet in practice, this method can create the customization and version-management headaches that portability is meant to eliminate. Another conditional code approach, implementing if-else conditions to select a processor-specific execution path at runtime, results in both unmanageable code and wasted CPU cycles.

> "One proven technique is to avoid making assumptions about integer and pointer sizes"

All told, it's better to stick to ANSI C and to use truly platform-independent data structures and access methods as much as possible to work around compiler- and platform-specific issues.

In the process of creating the eXtreme-DB in-memory embedded database at McObject (where I work), we developed several techniques that are useful for any developer seeking to write highly portable, maintainable, and efficient embedded code. Some of these techniques apply to embedded systems portability generally, but are particularly important for data management. In many cases, an embedded application's database is its most complex component, and getting it right the first time (by implementing highly portable code) saves programmer-months down the road. Other techniques I present here, such as building lightweight database synchronization based on a user-mode spinlock, constitute specific key building blocks for portable embedded systems databases.

Word Sizes

One proven technique is to avoid making assumptions about integer and pointer sizes. Defining the sizes of all base types used throughout the database engine code, and putting theses *typedefs* in a separate header file, makes it much easier to change them when moving the code from one platform to another or even using a different compiler for the same hardware platform; see Listing One.

Defining a pointer size as a *sizeof(void*)* and using the definition to calculate memory layout offsets or using it in pointer arithmetic expressions avoids surprises when moving to a platform such as ZiLOG eZ80 with 3-byte pointers:

```
#define PTRSIZE sizeof(void *)
```

The *void** type is guaranteed to have enough bits to hold a pointer to any data object or to a function.

Data Alignment

Data alignment can be a portability killer. For instance, on various hardware architectures a 4-byte integer may start at any address, or start only at an even address, or start only at a multiple-of-four address. In particular, a structure could have its elements at different offsets on different architectures, even if the element is the same size. To compensate, our in-memory data layout requires data object allocation to start from a given position, and aligns elements via platform-independent macros. Listing Two aligns the position of the data object (*pos*) at a 4-byte boundary.

Another alignment-related pitfall is that, on some processors (such as SPARC), all data types must be aligned on their natural boundaries. Using Standard C data types, integers are aligned as follows:

- *short* integers are aligned on 16-bit boundaries.
- *int* integers are aligned on 32-bit boundaries.
- *long* integers are aligned on either 32-bit boundaries or 64-bit boundaries, depending on whether the data model of the kernel is 64-bit or 32-bit.
- *long long* integers are aligned on 64-bit boundaries.

Usually, the compiler handles these alignment issues and aligns the variables automatically; see Listing Three. But redefining the way a variable or a structure element is accessed, while possible and sometimes desirable, can be risky. For example, consider the declaration in Listing Four of an object handle (assuming the data object size is *N* bytes). Such opaque handle declarations are commonly used to hide data object representation details from applications that access the data object with an interface function, using the handle merely as the object identifier as shown in Listing Five. Because *d* is a byte array, the address is not memory aligned. The handle is further used as an identifier of the object to the library:

```
void* function (  appData *handle);
```

Furthermore, internally the library "knows" about the detail behind the handle and declares the object as a structure with the elements defined as short integers, long integers, references, and so on; see Listing Six.

Accessing object elements leads to a bus error because they are not correctly aligned. To avoid the problem, we declare in Listing Seven the object handle as an array of operands of the maximum size (as opposed to a byte array). In this case, the compiler automatically aligns the operands to their natural boundaries, preventing the bus error.

Word Endianness

Byte order is the way the processor stores multibyte numbers in memory. Big-endian machines, such as Motorola 68k and SPARC, store the byte with the highest value digits at the lowest address while Little-endian machines (Intel 80x86) store it at the highest address. Furthermore, some CPUs can toggle between Big- and Little-endian by setting a processor register to the desired endian-architecture (IBM PowerPC, MIPS, and Intel Itanium offer this flexibility). Therefore, code that depends on a particular orientation of bits in a data object is inherently nonportable and should be avoided. Portable, endian-neutral code should make no assumptions of the underlying processor architecture, instead wrapping the access to data and memory structures with a set of interfaces implemented via processor-independent macros, which automatically compile the code for a particular architecture.

Furthermore, a few simple rules help keep the internal data access interfaces portable across different CPU architectures.

- Access data types natively; for instance, read an *int* as an integer number as opposed to reading 4 bytes.
- Always read/write byte arrays as byte arrays instead of different data types.
- Bit fields defined across byte boundaries or smaller than 8 bits are nonportable. When necessary, to access a bit field that is not on byte boundaries, access the entire byte and use bit masks to obtain the desired bits.
- Pointer casts should be used with care. In endian-neural code, casting pointers that change the size of the pointed-to data must be avoided. For example, casting a pointer to a 32-bit value 0x12345678 to a byte pointer would point to 0x12 on a Big-endian and to 0x78 on a Little-endian machine.

Compiler Differences

Compiler differences often play a significant role in embedded systems portability. Although many embedded environments are said to conform to ANSI Standards, it is well known that in practice, many do not. These nonconformance cases are politely called "limitations." For example, although required by the Standard, some older compilers recognize *void*, but don't recognize *void**. It is difficult to know in advance whether a compiler is in fact a strict ANSI C compiler, but it is very important for any portable code to follow the Standard. Many compilers allow extensions; however, even common extensions can lead to portability problems. In our development, we have come across several issues worth mentioning to avoid compiler-dependent problems.

When *char* types are used in expressions, some compilers treat them as unsigned, while others treat them as signed. Therefore, portable code requires that *char* variables be explicitly cast when used in expressions; see Listing Eight.

Some compilers cannot initialize auto-aggregate types. For example, Listing Nine may not be allowed by the compiler. The most portable solution is to add code that performs initialization, as in Listing Ten.

C-Runtime Library

Databases in nonembedded settings make extensive use of the C runtime. However, embedded systems developers commonly avoid using the C runtime, to reduce memory footprint. In addition, in some embedded environments, C-runtime functions,

such as dynamic memory allocations/deallocations (*malloc()/free()*), are implemented so poorly as to be virtually useless.

An alternative, implementing the necessary C-runtime functionality within the database runtime itself, reduces memory overhead and increases portability. For main-memory databases, implementing dynamic memory management through the database runtime becomes vitally important because these engines' functionality and performance are based on the efficiency of memory-oriented algorithms. We incorporate a number of portable embedded memory-management components that neither rely on OS-specific, low-level memory-management primitives, nor make any fundamental assumptions about the underlying hardware architecture. Each of the memory managers employs its own algorithms, and is used by the database runtime to accomplish a specific task.

- A dynamic memory allocator provides functionality equivalent to the standard C runtime library functions *malloc(), calloc(), free(),* and *realloc()* according to the POSIX Standard. The heap allocator is used extensively by the database runtime, but also can be used by applications.
- Another memory manager is a comprehensive data layout page manager that implements an allocation strategy adapted to the database runtime requirements. Special care is taken to avoid introducing unnecessary performance overhead associated with multithreaded access to the managed memory pools.
- A simple and fast single-threaded memory manager is used while parsing SQL query statements at runtime, and so on.

Synchronization

Databases must provide concurrent access across multiple simultaneously running tasks. Regardless of the database locking policies (optimistic or pessimistic, record level or table level, and the like), this mechanism is usually based on kernel synchronization objects, such as semaphores, provided by the underlying OS. While each operating system provides very similar basic synchronization objects, they do so with considerably different syntax and usage, making it nontrivial to write portable multithreaded synchronization code. In addition, an embedded systems database must strive to minimize the expense associated with acquiring kernel-level objects. Operating-system semaphores and mutexes are usually too expensive, in performance terms, to be used in embedded settings.

In our case, the solution was to build up the database runtime synchronization mechanism based on a simple synchronization primitive—the test-and-set method—that is available on most hardware architectures. Foregoing the kernel for a hardware-based mechanism reduces overhead and increases portability. All we must do is port three functions to a specific target. This approach can also be used for ultra low-overhead embedded systems where no operating system is present (hence, no kernel-based synchronization mechanism is available). Furthermore, the performance of the test-and-set "latch" in Listing Eleven remains the same regardless of the target operating system and depends only on the actual target's CPU speed. Listing Twelve(a) provides implementations for Win32, Listing Twelve(b) the Sun SPARC platform, and Listing Twelve(c) is the Green Hills INTEGRITY OS.

The concept of mutual exclusion is crucial in database development—it provides a foundation for the ACID properties that guarantee safe sharing of data. The synchronization approach just discussed slants toward the assumption that for embedded systems databases, it is often more efficient to poll for the availability of a lock rather than allow fair preemption of the task accessing the shared database.

It is important to note that even though this approach is portable in the sense that it provides consistent performance of the synchronization mechanism over multiple operating systems and targets, it does not protect against the "starvation" of tasks waiting for, but not getting, access to the data. Also, provisions must be made for the database system to clean itself up if the task holding the lock unexpectedly dies, so that other tasks in line for the spin-lock do not wait eternally. In any case, embedded data management is often built entirely in memory, generally requires a low number of simultaneous transactions, and the transactions themselves are short in duration. Therefore, the chances of a resource conflict are low and the task's wait to gain access to data is generally shorter than the time needed for a context switch.

Nonportable Features

While replacing C runtime library functionality and memory managers, and implementing custom synchronization primitives lead to greater data-management code portability, sometimes it is not possible or practical to overload the database with functionality—such as network communications or filesystem operations—that belongs to the operating system. A solution is to not use these services directly, or recreate them in the database, but instead create an abstraction of them that is used throughout the database engine code. The actual implementation of the service is delegated to the application. This allows hooking up service implementations without changing the core engine code, which again contributes to portability.

For example, data-management solutions often include online backup/restore features that, by their nature, require file system or network interaction. Creating an abstraction of stream-based read and write operations, and using this abstraction layer within the database runtime during backup, allows the database to implement the backup/restore logic while staying independent of the actual I/O implementation. At the same time, this approach allows a file-based, socket-based, or other custom stream-based transport to be plugged in with no changes needed to the database runtime. Listing Thirteen illustrates such a plug-in interface. The application needs to implement the actual read-from-stream/write-to-stream functionality; see Listing Fourteen.

Another example of the database "outsourcing" services to the application involves network communications. Embedded databases often must provide a way to replicate the data between several databases over a network. Embedded settings always demand highly configurable and often deterministic communication that is achieved using a great variety of media access protocols and transports. Thus, as a practical matter, a database should be able to adopt the communication protocol used for any given embedded application, regardless of the underlying hardware or the operating system. Instead of communicating directly with the transport or a protocol, a database runtime goes through a thin abstraction layer that provides a notion of a "communication channel." Like the backup/restore interfaces, the network communication channel can also be implemented via a stream-based transport; see Listing Fifteen. The database uses a set of API functions that provide the ability to initiate and close the channel, send and receive the data, and so on.

Conclusion

By following general rules of developing portable code—such as using Standard C and avoiding assumptions about hardware related parameters—you can greatly simplify the reuse of your data-management code in new environments. And new approaches to implementing standard database services, such as those presented here, can ensure that the old concept of a database delivers the portability, performance, and low-resource consumption demanded for embedded systems.

DDJ

(Listings begin on page 54.)

Listing One

```
#ifndef BASE_TYPES_DEFINED

typedef unsigned char   uint1;
typedef unsigned short  uint2;
typedef unsigned int    uint4;
typedef signed char     int1;
typedef short           int2;
typedef int             int4;

#endif
```

Listing Two

```
#define ALIGNEDPOS(pos, align) ( ((pos) + (align)-1) & ~((align)-1) )
pos = ALIGNEDPOS(pos, 4);
```

Listing Three

```
char c;
//(padding)
long  l;        - the address is aligned
```

Listing Four

```
#define handle_size      N
typedef uint1 hobject    [handle_size ];
```

Listing Five

```
typedef struct  appData_ { hobject h; } appData;
char c;
appData d;    /* d is not aligned */
```

Listing Six

```
typedef struct objhandle_t_
{
  ...
  obj_h          po;
  ...
  uint4          mo;
  uint2          code;
  ...
} objhandle_t;
```

Listing Seven

```
#define handle_size      N
#define handle_size_w
((( handle_size + (sizeof(void*) -1)) & ~(sizeof (void*) -1)) / sizeof(void*));

typedef void * hobject [handle_size_w ];
```

Listing Eight

```
#if defined( CFG_CHAR_CMP_SIGNED )
#define CMPCHARS(c1,c2) ((int)(signed char)(c1)-(int)(signed char)(c2) )
#elif defined( CFG_CHAR_CMP_UNSIGNED )
#define CMPCHARS(c1,c2) ((int)(unsigned char)(c1)-(int)(unsigned char)(c2) )
#else
#define CMPCHARS(c1,c2) ( (int)(char)(c1) - (int)(char)(c2) )
#endif
```

Listing Nine

```
struct S { int i; int j; };
S s = {3,4};
```

Listing Ten

```
struct S { int i; int j; };
S s;
s.i = 3; s.j = 4;
```

Listing Eleven

```
/* this is the TAS (test-and-set) latch template*/
void sys_yield()
{
  /* relinquish control to another thread */
}
void sys_delay(int msec)
{
  /* sleep */
}
int  sys_testandset( /*volatile*/ long * p_spinlock)
{
  /* The spinlock size is up to a long ;
   * This function performs the atomic swap (1, *p_spinlock) and returns
   * the previous spinlock value as an integer, which could be 1 or 0
  */
}
```

Listing Twelve

(a)

```
#ifndef SYS_WIN32_H__
#define SYS_WIN32_H__

/* sys.h definitions for WIN32 */

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <process.h>

#define sys_yield() SleepEx(0,1) /*yield()*/
#define sys_delay(msec) SleepEx(msec,1)
#define sys_testandset(ptr) InterlockedExchange(ptr,1)

#endif  /* SYS_WIN32_H__ */
```

(b)

```
#ifndef SYS_SOL_H__
#define SYS_SOL_H__

/* sys.h definitions for Solaris */

#include <sys/time.h>
#include <unistd.h>
#include <sched.h>

int  sys_testandset( /*volatile*/ long * p_spinlock)
{
  register char result = 1;
  volatile char *spinlock = ( volatile char * ) p_spinlock;
    __asm__ __volatile__(
        "   ldstub   [%2], %0    \n"
:       "=r"(result), "=m"(*spinlock)
:       "r"(spinlock));
    return (int) result;
}
void sys_yield()
{
  sched_yield();
}
void sys_delay(int msec)
{ /* */ }
```

(c)

```
#ifndef SYS_GHSI_H__
#define SYS_GHSI_H__

/* sys.h definitions for Green Hills Integrity OS */
#include <INTEGRITY.h>

void sys_yield()
{
  Yield();
}
void sys_delay(int msec)
{
}
int  sys_testandset(long * p_spinlock)
{
  return ! ( Success == TestAndSet(p_spinlock, 0, 1) );
}
```

Listing Thirteen

```
/* abstraction of write and read stream interfaces;
 * a stream handle is a pointer to the implementation-specific data
*/
typedef int (*stream_write)
   ( void *stream_handle, const void * from, unsigned nbytes);
typedef int (*stream_read)
   ( void *stream_handle, /*OUT*/ void * to, unsigned max_nbytes);
/* backup the database content to the output stream */
RETCODE db_backup
   ( void * stream_handle, stream_write output_stream_writer, void * app_data);
/* restore the database from input stream */
RETCODE  db_load
    ( void * stream_handle, stream_read input_stream_reader, void *app_data);
```

Listing Fourteen

```
int file_writer(void *stream_handle, const void * from, unsigned nbytes)
{
  FILE *f = (FILE*)stream_handle;
  int nbytes = fwrite(from,1,nbytes,f);
  return nbytes;
}
int file_reader(void *stream_handle, void * to, unsigned max_nbytes)
{
  FILE *f = (FILE*)stream_handle;
  int nbytes = fread(to,1,max_nbytes,f);
  return nbytes;
}
```

Listing Fifteen

```
#define channel_h void*
typedef int (*xstream_write)(channel_h ch, const void * from,
                             unsigned nbytes, void * app_data);
typedef int (*xstream_read) (channel_h ch, void * to,
                             unsigned max_nbytes, void* app_data);
typedef struct {
  xstream_write fsend;
  xstream_read  frecv;
  ...
 } channel_t, *channel_h;
```

DDJ

Performance Monitoring with PAPI

Using the Performance Application Programming Interface

PHILIP MUCCI
WITH CONTRIBUTIONS FROM
NILS SMEDS AND PER EKMAN

Contrary to widely held misconceptions, performance *is* portable. Virtually every optimization effort initially involves tuning for architectural features found on most nonvector processors. The code is modified to improve data reuse in the caches, reduce address misses in the TLB, and to prevent register spills to memory, all of which can cause stalls in the processor's pipeline. But the questions often arise: Why should we optimize code at all? Why not just use the best available compiler options and move on? If the code runs too slowly, why not just upgrade our hardware and let Moore's Law do the work for us? There are many answers to these questions, the simplest being that the performance of most applications depends as much on the bandwidth of the memory subsystem as the core clock speed of the processor. While we do see processor megahertz continuing to double every 18 months, DRAM speeds see 10 percent improvement in bandwidth at best. Another reason to optimize has to do with the average performance of the user's hardware. Many applications, such as an MP3 player or video conferencing package, have a certain amount of work to do in a finite amount of time. Considering that most of the user community is running computers that are at least a generation or two behind, optimizing this software guarantees that more people will be able to run your code.

In the high-performance computing (HPC) community, optimization is of critical importance as it directly translates to better science. At HPC centers, computer time is "sold" to various parties of each institution. Take, for example, the National Energy Research Scientific Computing Center (NERSC)—the computer center at Lawrence Berkeley National Laboratory (http://www.lbl.gov/). The center runs a variety of large machines and sells time to each of the different divisions of the lab, as well as other government and academic institutions around the country. On these systems, 99 percent of the workload is for scientific simulation for running virtual experiments beyond physical or economical constraints. These applications range from the simulation of self-sustaining fusion reactors to climate modeling of the entire Earth down to a resolution of a few kilometers. The compute time of these models is always dependent on the resolution of the simulation. To fit within budget, experimenters must choose an appropriate resolution that resolves the phenomena of interest, yet still remains within their allocation. Optimization of these applications can result in a tremendous savings of compute time, especially when you consider

> "The hardest part of the optimization process is understanding the interaction of the system architecture, operating system, compiler, and runtime system"

Philip is the original author and technical lead for PAPI. He is a research consultant for the Innovative Computing Laboratory at the University of Tennessee, Knoxville. Nils and Per are computer scientists at the Center for Parallel Computers (PDC) at the Royal Institute of Technology in Stockholm, Sweden, and are regular contributors to the PAPI project. They can be contacted at mucci@cs.utk.edu, smeds@pdc.kth.se, and pek@pdc.kth.se, respectively.

Open-Source Performance Analysis Tools

The PerfSuite tool is from the National Center for Supercomputing Applications (http://perfsuite.ncsa.uiuc.edu/). PerfSuite includes a number of tools and libraries for performance analysis of serial, MPI, and OpenMP applications. Of initial interest to most users is the PSRUN tool that, when run on an application, generates textual output of raw performance counter data as well as a wealth of derived statistics. Figure 1 is an example of a run of the genIDLEST application. PSRUN can also be used to generate flat statistical profiles by file, function, and source line of an application.

The HPCToolkit, from the Center for High Performance Software Research at Rice University (http://www.hipersoft.rice.edu/hpctoolkit/), includes hpcrun—a command-line tool to generate flat statistical profiles to be visualized with a Java-based GUI called "hpcview" as shown in Figure 2. HPCToolkit is capable of generating and aggregating multiple simultaneous profiles such that for a postanalysis, line-by-line profiles can be generated from multiple hardware metrics. Like PerfSuite, HPCToolkit supports serial and parallel execution models.

Often after an initial performance analysis, users wish to narrow the focus of their examination. This may involve making detailed measurements of regions of code and associating them with an application event, such as updates of a cell in a grid. TAU, the "Tuning and Analysis Utilities" from the University of Oregon, is such a tool (http://www.cs.uoregon.edu/research/paracomp/tau/tautools/). TAU is a comprehensive environment for performance analysis that includes modules for both source and binary instrumentation, experiment management of performance data, and scalable GUIs suitable for visualization of highly parallel programs. TAU runs just about everywhere and can even function without PAPI being available (although only time-based metrics are available without PAPI). Figure 3 is a TAU display of the performance of a 16-process MPI program.

—P.M., N.S., and P.E.

```
PerfSuite 1.0 summary for execution of gen.ver2.3.inte (PID=9867, domain=user)

        Based on 800 MHz -1
        GenuineIntel 0 CPU
        CPU revision 6.000

Event Counter Name                                          Counter Value
================================================================================

  0 Conditional branch instructions mispredicted.....       3006093956
  1 Conditional branch instructions correctly predicted.....  32974709880
  2 Conditional branch instructions taken.....              26952022279
  3 Floating point instructions.....                        44525980237
  4 Total cycles.....                                       353262206234
  5 Instructions completed.....                             489764680025
  6 Level 1 data cache accesses.....                        56390921533
  7 Level 1 data cache hits.....                            41911206947
  8 Level 1 data cache misses.....                          14615753570
  9 Level 1 load misses.....                                17611912424
 10 Level 1 cache misses.....                               17597248300
 11 Level 2 data cache accesses.....                        53158617899
 12 Level 2 data cache misses.....                          8440205387
 13 Level 2 data cache reads.....                           43528651785
 14 Level 2 data cache writes.....                          10240563775
 15 Level 2 load misses.....                                3615923337
 16 Level 2 store misses.....                               667575973
 17 Level 2 cache misses.....                               8529931717
 18 Level 3 data cache accesses.....                        3826843278
 19 Level 3 data cache hits.....                            2799591986
 20 Level 3 data cache misses.....                          999714206
 21 Level 3 data cache reads.....                           3573882130
 22 Level 3 data cache writes.....                          171800425
 23 Level 3 load misses.....                                944624814
 24 Level 3 store misses.....                               49427000
 25 Level 3 cache misses.....                               1024569375
 26 Load instructions.....                                  84907675686
 27 Load/store instructions completed.....                  95346092870
 28 Cycles Stalled Waiting for memory accesses.....         140032176122
 29 Store instructions.....                                 10267472354
 30 Cycles with no instruction issue.....                   67247126931
 31 Data translation lookaside buffer misses.....           8365029

Statistics
================================================================================
Graduated instructions/cycle.....                          1.386406
Graduated floating point instructions/cycle.....           0.126042
Graduated loads & stores/cycle.....                        0.269902
Graduated loads & stores/floating point instruction.....   2.141359
L1 Cache Line Reuse.....                                   5.523515
L2 Cache Line Reuse.....                                   0.731682
L3 Cache Line Reuse.....                                   7.442618
L1 Data Cache Hit Rate.....                                0.846708
L2 Data Cache Hit Rate.....                                0.422527
L3 Data Cache Hit Rate.....                                0.881553
% cycles w/no instruction issue.....                       19.036037
% cycles waiting for memory access.....                    39.639729
Correct branch predictions/branches taken.....             1.000000
MFLOPS.....                                                100.833839
```

Figure 1: *Output from the PSRUN tool of PerfSuite showing hardware events and derived metrics gathered via PAPI.*

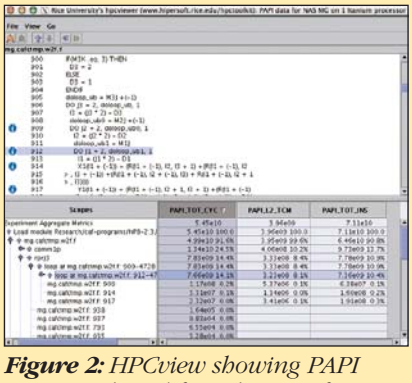


Figure 2: *HPCview showing PAPI events gathered from hpcrun for a triply nested loop.*

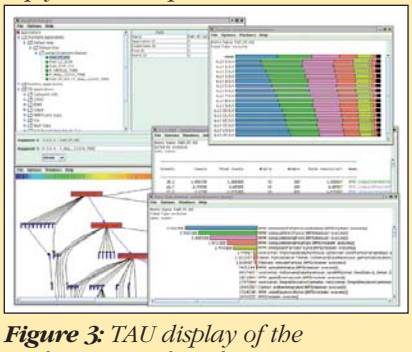


Figure 3: *TAU display of the performance of a 16-process MPI program with multiple PAPI metrics, subroutine breakdown, and call graph display.*

that these simulations are parallel codes; there may be thousands of processors executing similar code. A 30 percent decrease in runtime could allow for a 30 percent increase in the resolution of the simulation, possibly exposing subtleties not present in the former simulation. Such a difference directly translates into accelerating the pace of scientific discovery, and even (as in the case of car crash simulations or airplane wing design) result in saved lives.

However, the hardest part of the optimization process is understanding the interaction of the system architecture, operating system, compiler, and runtime system and how that affects the performance of the application. All these elements must be taken into consideration when attempting to optimize an application. Wouldn't it be nice if there was a way to reduce the prerequisite knowledge and experience by having the processor tell you exactly where and why your code was losing performance? Well, it turns out that such a method does exist— on-chip performance monitoring (PM) hardware found on almost every microprocessor in use today. The PM hardware consists of a small number of registers with connections to various other parts of the chip. Traditionally, this hardware was used exclusively for testing and verification. However, now the importance of the PM hardware has become widely understood, especially among the HPC community.

There are two methods of using the PM hardware— aggregate (direct) and statistical (indirect):

- Aggregate usage involves reading the counters before and after the execution of a region of code and recording the difference. This usage model permits explicit, highly accurate, fine-grained measurements. There are two subcases of aggregate counter usage: Summation of the data from multiple executions of an instrumented location, and trace generation, where the counter values are recorded for every execution of the instrumentation.
- The second method is statistical profiling: The PM hardware is set to generate an interrupt when a performance counter reaches a preset value. This interrupt carries with it important contextual information about the state of the processor at the time of the event. Specifically, it includes the program counter (PC), the text address at which the interrupt occurred. By populating a histogram with this data, users obtain a probabilistic distribution of PM interrupt events across the address space of the application. This kind of profiling facilitates a good high-level understanding of where and why the bottlenecks are occurring. For instance, the questions, "What code is responsible for most of the cache misses?" and "Where is the branch prediction hardware performing poorly?" can quickly be answered by generating a statistical profile.

In "Optimization Techniques" (*DDJ*, May 2004), Tim Kientzle describes how to use the real-time cycle counter (*rdtsc*) found on the x86 architecture. However, there are some problems with this technique. The first is due to the nature of a system designed for multiprocessing. No system can be considered quiet when examined from a sufficiently fine granularity. For example, the laptop on which this article is being written is busy servicing interrupts from a variety of sources, as well as delivering commands to the graphics coprocessor and the PCM audio chip. The consequence of this is that it is very likely that the cycle count from *rdtsc( )* includes a host of other unrelated events. On a desktop system, a stock Redhat 9 system with no "active" processes, the system (*vmstat*) reports about 120 interrupts and 50 context switches every 10 seconds. Doing cycle-by-cycle accounting of code is thus impossible— just one interrupt, or one context switch during a measurement interval, could mislead you in your quest for bottlenecks.

The next problem with timers is more severe. Timers don't tell you anything about why the hardware is behaving the way it is. The only way to understand the

system is to attempt to reconstruct the processor's pipeline as it executes your code. This is a virtually impossible task on today's advanced CPUs, even for experts with detailed knowledge of the processors microarchitecture. The last problem is one of portability and interface semantics. While the *rdtsc()* code segment works nicely in the Intel/Microsoft environment, it doesn't help when you migrate to another architecture, operating system, or even compiler!

These problems and more have been addressed with the development of the Performance Application Programming Interface (PAPI) library. PAPI (available at http://icl.cs.utk.edu/papi/) is intended to be completely portable from an API standpoint. That is, performance tools and instrumentation that work on one platform, seamlessly work with a recompile on another. While the interface is portable, the generated data is not. Data from the PM hardware rarely has the same semantics from vendor to vendor, and often changes from model to model. A cache miss on one platform may be measured entirely differently on another; even definitions as "simple" as an instruction can become blurred on modern architectures (consider x86 instructions versus x86 υops).

PAPI is implemented on a wide variety of architectures and operating systems. The current release, PAPI 3.0.7, is supported on the Cray T3E, X1, Sun UltraSparc/Solaris, Alpha/Tru64, IBM Power 604e,2,3,4/AIX, MIPS R10k/IRIX, IA64,IA32,x86_64/Linux, and Windows/IA32 (not P4). Wrappers for PAPI exist for C, C++, Fortran, Java, and Matlab.

To facilitate the development of portable performance tools, PAPI provides interfaces to get information about the execution environment. It also provides methods to obtain a complete listing of what PM events are available for monitoring. PAPI supports two types of events, preset and native. Preset events have a symbolic name associated with them that is the same for every processor supported by PAPI. Native events, on the other hand, provide a means to access every possible event on a particular platform, regardless of there being a predefined PAPI event name for it.

For preset events, users can issue a query to PAPI to find out if the event is present on the current platform. While the exact semantics of the event might be different from processor to processor, the name for the event is the same (for example, *PAPI_TOT_CYC* for elapsed CPU cycles). Native events are specific to each processor and have their own symbolic name (usually taken from the vendor's architecture manual or header file, should one exist). By their nature, native events are always present.

PAPI supports measurements per-thread; that is, each measurement only contains counts generated by the thread performing the PAPI calls. Each thread has its own PM context, and thus can use PAPI completely independently from other threads. This is achieved through the operating system, which saves/restores the performance counter registers just like the rest of the processor state at a context switch. Using a lazy save and restore scheme effectively reduces the additional instruction overhead to a few dozen cycles for the processes/threads that are using the performance monitoring hardware. Most operating systems (AIX, IRIX, Tru64, Unicos, Linux/IA64, HPUX, Solaris) have officially had this support for a significant time, motivated largely by the popularity of the PAPI library and associated tools. For Linux/IA32/x86_64, support exists in the form of a PerfCtr kernel patch (http://user.it.uu.se/~mikpe/linux/perfctr/), written by Mikael Pettersson of Uppsala University in Sweden. This patch has not yet been formally accepted in the main Linux 2.6 kernel tree. Users with concerns about stability and support should know that this patch has been installed and in use for many years at a majority of the U.S. Government's Linux clusters on the Top 500 list, not to mention numerous other computational intensive sites around the world. The notable exception to operating system support is Microsoft Windows. Due to the lack of support of preserving the state of performance counters across context switches, counting on Windows must be done system-wide, greatly reducing the usefulness of the PM for application development and tuning.

PAPI has two different library interfaces. The first is the high-level interface meant for use directly by application engineers. It consists of eight functions that make it easy to get started with PAPI. It provides start/read/stop functionality as well as quick and painless ways to get information, such as millions of floating-point operations per second (MFLOPS/S) and instructions per cycle. Listing One contains code that demonstrates a canonical performance problem—traversing memory with nonunit stride. We measure this code's performance using the PAPI high-level interface. This example uses PAPI presets. They are portable in name but might not be implemented on all platforms, and may in fact mean slightly different things on different platforms. Hardware restrictions also limit which events can be measured simultaneously. This simple example verifies that the hardware supports at least two simultaneous counters; it then starts the performance counters counting the preset events for L1 data cache misses (*PAPI_L1_DCM*) and the number of floating-point operations executed (*PAPI_FP_OPS*). Compiling and running this example on an Opteron (64-byte line size) with gcc 3.3.3 (-O -g) results in this output:

```
Total software flops = 41943040.000000
Total hardware flops = 42001076.000000
MFlop/s = 48.258228
L2 data cache misses is 12640205
```

Advanced PM Features

Traditionally, performance monitoring hardware counts the occurrence or duration of events. However, recent CPUs, such as the Intel Itanium2 and IBM Power4/5, include more advanced features. The Itanium2 contains such features as:

- Address and opcode matching. With address matching, the event counting can be constrained to events that are triggered by instructions that either occupy a certain address range in the code or reference data in a certain address range. The first case enables you to measure, for instance, the number of mispredicted branches within a procedure in a program. In the second case, you can measure things like all the cache misses caused by references to a particular array.

  Opcode matching constrains counting to events caused by instructions that match a given binary pattern. This allows the counting of instructions of a certain type or execution on a particular functional unit.
- Branch traces. A CPU with a branch trace buffer (BTB) can store the source and target addresses of branch instructions in the PMU as they happen. On the Itanium2, the branch trace buffer consists of a circular buffer of eight registers in which source and target addresses are stored. Address capturing can be constrained by the type of branch, whether it was correctly predicted and whether the branch was taken.
- Event addresses. With event address capturing, addresses and latencies associated with certain events are stored in Event Address Registers (EARs). This means that the PM hardware captures the data and instruction addresses that cause, for example, an L1 D-cache miss together with the latency (in cycles) of the associated memory access.

—P.M., N.S., and P.E.

```
/avail -e PAPI_FP_INS
Test case avail.c: Available events and hardware information.
-------------------------------------------------------------------------
Vendor string and code:                 AuthenticAMD (2)
Model string and code:                  AMD K8 Revision C (15)
CPU Revision:                           8.000000
CPU Megahertz:                          1993.626953
CPU's in this Node:                     4
Nodes in this System:                   1
Total CPU's:                            4
Number Hardware Counters:               4
Max Multiplex Counters:                 32
-------------------------------------------------------------------------
Event name:                             PAPI_FP_INS
Event Code:                             0x80000034
Number of Native Events:                1
Short Description:                      |FP instructions|
Long Description:                       |Floating point instructions|
 |Native Code[0]:                       0x40000002  FP_MULT_AND_ADD_PIPE|
 |Number of Register Values:            2|
 |Register[0]: 0xf                      P3 Ctr Mask|
 |Register[1]: 0x300                    P3 Ctr Code|
 |Native Event Description:             |Dispatched FPU ops - Revision B and later revisions -
                                         Speculative multiply and add pipe ops excluding junk ops|
```

Figure 4: *Output from the avail tool from the PAPI distribution.*

(continued from page 58)

There's almost one cache miss for every four floating-point operations—performance is bound to be poor. Switching the *for* loops so that the memory is accessed sequentially instead should give us better cache behavior; the result is:

```
Total software flops = 41943040.000000
Total hardware flops = 42027880.000000
MFlop/s = 234.481339
L2 data cache misses is 2799387
```

Performance has more than quadrupled and the cache misses have been reduced by 80 percent. Here we note that the number of hardware flops is different from run to run. Using "avail" (Figure 4) helps answer this question. Note that the output from avail has been edited for brevity. The *PAPI_FP_INS* preset is implemented as the vendor *FP_MULT_AND_ADD_PIPE* metric. This metric is speculative, meaning the hardware counter includes instructions that were not retired.

The low-level interface is designed for power users and tool designers. It consists of 53 functions that range from providing information about the processor and executable to advanced features like counter multiplexing, callbacks on counter overflow, and advanced statistical profiling modes. Counter multiplexing is a useful feature in situations where the PM hardware has a very limited number of registers. This limitation can be overcome by trading accuracy and granularity of the measurements for an increase in the measured number of events. By rapidly switching the contents of the PM's control registers during the course of a user's run, the appearance of a great number of PM registers is given to users. For more on advanced features, see the accompanying text box entitled "Advanced PM Features."

While PAPI provides the necessary infrastructure, the true power of PAPI lies in the tools that use it. There are a number of excellent open-source performance analysis tools available that have been implemented using PAPI. While a complete discussion of all these tools is beyond the scope of this article, we mention a few in the accompanying text box entitled "Open-Source Performance Analysis Tools." For information about these and other tools, please refer to the PAPI web page.

PAPI's goal is to expose real hardware performance information to users. By doing so, most of the guesswork regarding the root cause of a code's performance problem can be eliminated. PAPI does not solve algorithmic design issues or diagnose inefficient parallelization. It can, however, diagnose poor usage of the available processor resources—a problem that, before now, was largely intractable. As of now, PAPI is an ad hoc standard, taking into account the input of the tool development and performance engineering community. In the future, we hope to see PAPI evolve into a true open standard and to see the vendors ship native versions of PAPI with each new release of processor and operating system. In the meantime, with continued cooperation from industry, academia, and the research community, PAPI will continue to evolve and further drive the science of performance engineering.

DDJ

Listing One

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/time.h>
#include "papi.h"

#define MX 1024
#define NITER 20
#define MEGA 1000000
#define TOT_FLOPS (2*MX*MX*NITER)

double *ad[MX];

/* Get actual CPU time in seconds */
float gettime()
{
return((float)PAPI_get_virt_usec()*1000000.0);
}
int main ()
{
float t0, t1;
int iter, i, j;
int events[2] = {PAPI_L1_DCM, PAPI_FP_OPS }, ret;
long_long values[2];

if (PAPI_num_counters() < 2) {
   fprintf(stderr, "No hardware counters here, or PAPI not supported.\n");
   exit(1);
}
for (i = 0; i < MX; i++) {
    if ((ad[i] = malloc(sizeof(double)*MX)) == NULL) {
        fprintf(stderr,"malloc failed"\n);
        exit(1);
    }
}
for (j = 0; j < MX; j++) {
    for (i = 0; i < MX; i++) {
        ad[i][j] = 1.0/3.0; /* Initialize the data */
    }
}
t0 = gettime();
if ((ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
   fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(ret));
   exit(1);
}
for (iter = 0; iter < NITER; iter++) {
    for (j = 0; j < MX; j++) {
        for (i = 0; i < MX; i++) {
            ad[i][j] += ad[i][j] * 3.0;
        }
    }
}
if ((ret = PAPI_read_counters(values, 2)) != PAPI_OK) {
  fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(ret));
  exit(1);
}
t1 = gettime();

printf("Total software flops = %f\n",(float)TOT_FLOPS);
printf("Total hardware flops = %lld\n",(float)values[1]);
printf("MFlop/s = %f\n", (float)(TOT_FLOPS/MEGA)/(t1-t0));
printf("L2 data cache misses is %lld\n", values[0]);
}
```

DDJ

The Technical Report On C++ Library Extensions

The Standards committee is nearly done with TR1

MATTHEW H. AUSTERN

The C++ Standard Library is a large and successful project. The specification of the library is about 370 pages in length—longer than the parts of the C++ Standard that describe the core language itself (ISO, International Standard: Programming languages—C++. ISO/IEC 14882:2003(E), 2003). There are now dozens of books explaining how to use the Standard Library (I've listed a few in the references). Nor is this just the domain of standardization bureaucrats or textbook writers. The C++ Standard was finalized more than six years ago, every major C++ compiler ships with a complete Standard Library implementation, and millions of programmers are using them. If a C++ program was written within the last five years (and often even if it was written earlier), it's standard fare for it to use *std::string*, STL containers and algorithms, or IOStreams.

Matthew is a software engineer at Google, chair of the C++ Standardization Committee's Library Working Group, project editor of the Technical Report on C++ Library Extensions, and author of Generic Programming and the STL. *He can be contacted at matt@lafstern.org.*

But if we look at it a little differently, the Standard Library doesn't seem so large after all. Granted, it's much larger than the C Standard Library (all of the 1990 C Library is included by reference!) and it includes an extensive collection of containers and algorithms (while the C Library has only qsort and bsearch). However, most of the problem areas the C++ Standard Library addresses—I/O and localization, character string manipulation, memory allocation, math functions—are in the C library as well. How does that compare to what other languages provide in their standard libraries?

The Perl, Python, and Java standard libraries include facilities for parsing XML, performing regular expression searches, manipulating image files, and sending data over the network. From that perspective, the C++ Standard Library looks like it's only the beginning.

And everyone in the Standards committee knows that, of course. When the Standard was finished in 1998, we didn't include everything that might have been a good idea—we only included what we could. Many potentially good ideas were left out for lack of time, lack of consensus, lack of implementation experience, or sometimes just lack of imagination. As Bjarne Stroustrup stated in the "The Design of C++0x" (*C/C++ Users Journal,* May 2005), the next C++ Standard will see far more emphasis on library evolution than core language changes. Within the Standards committee, wishing for a larger Standard Library is no more controversial than wishing for world peace.

The question isn't whether we need a more comprehensive library, but how we get one. Standards committees are good at evaluating ideas and working through corner cases, but they're slow, and they're poor at coming up with the ideas in the first place. "Design by committee," including design by Standards committee, is rarely a success.

"The question isn't whether we need a more comprehensive library, but how we get one"

Historically, members of the Standards committee came up with two answers. At the 1998 Santa Cruz committee meeting, Beman Dawes, then chair of the committee's Library Working Group, proposed that members of the C++ community work together informally to design and implement new library components, with the goal of advancing the state of the art and eventually building practice for a more extensive Standard Library. That proposal was wildly successful. It's what eventually became Boost (http://www.boost.org/).

But that still wasn't a complete answer. How does this informal library development, at Boost and elsewhere, get into something as formal as an official ISO Standard? At the 2001 Copenhagen meeting, Nico Josuttis and the rest of the German delegation came up with the second part—that we needed something in between the freedom of individual library development and the formality of an official ISO Standard. In 2001, a new revision of the C++ Standard seemed very far away, but many committee members were already working on libraries that they hoped could be part of that new Standard.

The fundamental issue is that, for obvious reasons, library development can go a lot faster than language development. The Standards committee decided to recognize that fact, and to start working on library extensions even before work on a new Standard began. We couldn't publish an official Standard on new library components, but what we could do, and what we did decide to do, was publish the work on library extensions in the form of a technical report. This report isn't "normative," in the jargon of standards bureaucracy, which means that compiler vendors don't have to include any of these new libraries to conform to the Standard. What it does is provide guidance to those vendors who do want to include them. In the meantime it gives us a head start on describing new library components in "standardese," gives us experience implementing and using new libraries, and will eventually make it easier to incorporate them into the next official Standard.

The idea of a library extensions technical report was first proposed at the 2001 Copenhagen meeting. The committee's library working group (which I chair) began discussing specific ideas for new library extensions at the next meeting and started to solicit proposals. The first extension proposals were accepted at the 2002 Santa Cruz meeting, and the last at the 2003 Kona meeting. Since then, the committee has been filling gaps, and fixing bugs and corner cases. At this writing, the C++ Standards committee has finalized the Proposed Draft Technical Report on C++ Library Extensions, ISO/IEC PDTR 19768, and forwarded it to our parent body. Most of the remaining steps are bureaucratic. There may be some minor changes before it is officially finalized, but the substantive work is now essentially complete; see http://www.open-std.org/jtc1/sc22/wg21/.

If you do download the technical report (TR) and read through it, you'll find that it looks a lot like the library part of the C++ Standard. Like the Standard, it contains header synopses; the Standard describes the contents of headers, such as <iterator> and <iostream>, like the TR describes the contents of headers, such as <tuple> and <regex>. (In some cases, such as <utility> and <functional>, the TR describes additions to existing headers.) Like the Standard, it contains classes and functions and type requirements. The style of specification in the TR is deliberately similar to the style in the Standard, because it is expected that much of what's in the TR will eventually become part of a future Standard.

There is one obvious difference between what's in the Standard and what's in the TR—the Standard Library defines all of its names within namespace *std,* but the TR defines its names within *std::tr1*. Why *std::tr1* instead of just *std::tr*? For the obvious reason: Someday there may be a namespace *std::tr2*! For similar reasons, instead of spelling out "Technical Report on C++ Library Extensions," most people just say "TR1." Work on new Standard Library components has started. It hasn't ended.

The fact that TR1 components aren't defined within namespace *std*, however, is a reminder of the real difference—as official as it looks, the TR isn't a standard. Everything in it, by definition, is experimental; it's there to get user and implementor experience. The expectation is that most of what's in TR1 will make it into the "C++0x" Standard (the next official version of the C++ Standard), but it's possible, once we've got more real-world experience, that there will still be some big changes along the way. So, while there's a lot in the TR that makes C++ programming easier and more fun, and while it gives you a taste of what the next Standard will look like, be aware that using the libraries defined in the TR means living on the bleeding edge.

With that caveat in mind, let's look at some of the high points.

Taking the STL Further: Containers

The STL, originally written by Alex Stepanov, Dave Musser, and Meng Lee, was the most innovative part of the C++ Standard Library (see Alexander Stepanov and Meng Lee, "The Standard Template Library," HP Technical Report HPL-95-11 (R.1), 1995). Generic STL algorithms, such as *sort* and *random_shuffle*, can operate on objects stored in STL containers, such as *vector* and *deque*, on objects stored in built-in arrays, and in data structures that haven't even been written yet. The only requirement is that these data structures provide access to their elements via iterators. Conversely, new algorithms, provided that they access the objects they operate on only through the iterator interface, can work with all existing and yet-to-exist STL data structures. For additional flexibility, STL algorithms parameterize some of their behavior in terms of "function objects" that can be invoked using function call syntax. For example, *find_if*, which searches for a particular element in a range, takes a function object argument that determines whether its argument matches the search, and *sort* takes a function object argument that determines whether one object is less than another.

The STL was designed to be extensible, and people immediately extended it. Some of the most obvious gaps to be filled were in the containers. The STL includes three containers for variable-sized sequences of elements: *vector, list,* and *deque*. It also includes set and dictionary classes based on balanced trees: the "associative containers" *set, map, multiset,* and *multimap*. A common reaction to that list is to see an obvious omission—hash tables. Most other languages, including Perl, Python, and Java, have hash-based dictionaries. Why doesn't C++?

The only real reason was historical. There was a proposal to add hash tables to the STL as early as 1995 (see Javier Barreiro, Robert Fraley, and David R. Musser, "Hash Tables for the Standard Template Library," X3J16/94-0218 and WG21/N0605, 1995), but that was already too late to make it into the Standard. Individual vendors filled that gap. Most major library implementations include some form of hash tables, including the Dinkumware standard library that ships with Microsoft Visual C++ and the GNU libstdc++ that ships with GCC. Something that's provided by different vendors in not-quite-compatible versions is a natural candidate for standardization, so the technical report, at last, includes hash tables.

TR1 hash tables have many specialized features, but in simple cases they're used much the same way as the standard associative containers. In Listing One, the name of this class might give you pause. Why is it called *unordered_map*, when all previous implementations used the name *hash_map*? Unfortunately, that history is the biggest reason for using a different name. It's impossible to make TR1 hash tables compatible with all previous STL hash table implementations because they aren't compatible with each other. Since the committee had to use a different interface, it felt that the right decision was to use a different name to go with it.

The unordered associative containers, like all STL containers, are homogeneous. All of the elements in an *std::vector<T>,* or an *std::trl::unordered_set<T>,* have the same type. But the Standard also has one heterogeneous container: *std::pair<T, U>,* which contains exactly two objects that may be of different types. Pairs are useful whenever two things need to be packaged together. One common use is for

functions that return multiple values. For example, the associative containers *set* and *map* (and *unordered_set* and *unordered_map*) have a version of insert whose return type is *pair<iterator, bool>*. The second part of the return value is a flag that tells you whether you did actually insert a new element or whether there was already one there with the same key, and the first part points to either the preexisting element or the newly inserted one.

Pairs are useful for packaging multiple values, so long as "multiple" means two. But that seems like an arbitrary restriction. Just as functions sometimes need to return more than one value, so they sometimes need to return more than two. In mathematics, an *n*-tuple is an ordered collection of *n* values; pairs are the special case where *n* is restricted to be 2. TR1 introduces a new heterogeneous container template, *std::tr1::tuple*, which removes that restriction.

Implementing *tuple* requires some sophisticated programming techniques, but using it couldn't be simpler. As with *pair*, you just supply the types as template parameters—except that with *pair* you supply exactly two template parameters, and with *tuple* you supply whatever number you like. (Up to some limit, but the limit should be large. On all implementations that I know of, it's at least 10.) For example:

```
#include <tr1/tuple>
#include <string>

using namespace std;
using namespace std::tr1;
 ...
tuple<int, char, string> t = make_tuple(1, 'a', "xyz");
```

If a function returns multiple values as a *tuple*, then, of course, one way to get those values is one at a time, the same as with a pair:

```
tuple<int, int, int> tmp = foo();
int x = get<0>(tmp);
int y = get<1>(tmp);
int z = get<2>(tmp);
```

The syntax is a little different than *pair*'s *first* and *second* members, but the idea is similar. With *tuple*s, however, there's an easier way—you don't even need that temporary variable. Instead, you can just write:

```
tie(x, y, z) = foo();
```

and let the library automatically handle all of this packing/unpacking.

Today, functions that need to return multiple values usually either pass in multiple reference parameters, or else define some ad hoc class to serve as a return type (think *div_t*). Now that we have *tuple*, which provides a dramatic improvement in usability, these clumsy workarounds might disappear.

Infrastructure: Smart Pointers and Wrappers

One reason *tuple* is useful is the same reason *string* is useful—it's a primitive that higher level libraries, including other parts of the Standard Library, can use in their own interfaces. What's important is that these types are a common vocabulary, so that two parts of a program, written independently of each other, can use them to communicate. The library extension technical report adds a number of other useful utility components of this nature.

> "One problem that appears in most programs is managing resource lifetimes"

One problem that appears in most programs is managing resource lifetimes. If you allocate memory or open a network socket, when does that memory get deallocated and when does the socket get closed? Two common solutions to that problem are automatic variables ("resource acquisition is initialization," or RAII) and garbage collection. Both solutions are useful and important, and every C++ programmer should be familiar with them. However, neither is appropriate in every circumstance. RAII works best when resource lifetime is statically determined and tied to the program's lexical structure, while garbage collection works better for memory than for other kinds of resources, and in any case is sometimes overkill. A third alternative, one that has been reinvented many times, is reference-counted smart pointers.

The basic idea behind reference-counted pointers is straightforward: Instead of trafficking in raw pointers of type *T**, programs can use some kind of wrapper class that "looks like" a pointer. Just as with an ordinary pointer, you can dereference a smart pointer to access the *T* object it points to, or, more commonly, you can use the -> operator to access one of that object's members. The difference is that the wrapper class can instrument its basic operations, its constructors and destructor and assignment operators, so that it can keep track of how many owners a particular object has.

The TR1 reference-counted pointer class is *shared_ptr*. In the simplest case, it works just like the standard *auto_ptr* class—you create an object the usual way with *new*, and use it to initialize the *shared_ptr*. If there aren't any other *shared_ptr* instances that refer to the *shared_ptr*, then the object is destroyed when the *shared_ptr* goes out of scope.

What's more interesting is what happens if there are other instances that point to the same object—it doesn't get destroyed until the last instance that refers to it goes away. You can confirm this by doing a simple test with a class that logs its constructors and destructors. In Listing Two, the two pointers, *p1* and *p2,* both point to the same *A* object, and both of those pointers are destroyed when they go out of scope. But *shared_ptr*'s destructor keeps track of that, so it doesn't destroy the *A* object until the last reference to it disappears.

Naturally, real examples are more complicated than this test case. You can also assign *shared_ptr*s to global variables, pass them to and return them from functions, and put them in STL containers. A *vector<shared_ptr<my_class> >* is one of the most convenient ways of managing containers of polymorphic objects (see my article "Containers of Pointers," *C/C++ Users Journal*, October 2001).

Providing *shared_ptr* as part of TR1 has two major benefits.

- As with *tuple* and *string*, it gives programs a common vocabulary: If I want to write a function that returns a pointer to dynamically allocated memory, I can use a *shared_ptr* as my return type and be confident that the clients of my function will have *shared_ptr* available. If I had written my own custom reference-counting class, that wouldn't have been true.
- Second (and perhaps more importantly), *shared_ptr* works. That's not as trivial as it might seem! Many people have written reference-counting smart pointer classes but many fewer people have written ones that get all the corner cases right—especially in a multithreaded environment. Classes such as *shared_ptr* are surprisingly easy to get wrong, so you definitely want an implementation that has been well tested and that has seen lots of user experience.

Reference-counted pointers don't completely remove the possibility of resource management bugs; some discipline by programmers is needed. There are two potential problems. First, suppose that two objects are pointing to each other. If *x* holds a *shared_ptr* to *y,* and *y* holds a *shared_ptr* to *x*, then neither reference count can ever drop to zero even if nothing else in the program points to either *x* or *y*. They form a cycle, and will eventually cause a memory leak. Second, suppose that you're mixing memory-management policies, and that you have both a *shared_ptr* and a regular pointer to the same object:

```
my_class* p1 = new my_class;
shared_ptr<my_class> p2(p1)
...
```

If *p1* outlives the last *shared_ptr* that's a copy of *p2,* then *p1* becomes a dangling pointer— and ends up pointing to an object that has already been destroyed. Trying to dereference *p1* will probably make your program crash. Some smart pointer libraries try to prevent this by making it impossible to access a smart pointer's underlying raw pointer, but *shared_ptr* doesn't. In my opinion, that was the right design decision. A general-purpose smart pointer class has to expose an underlying pointer somehow (otherwise *operator->* can't work), and throwing up artificial syntactic barriers just makes legitimate uses harder.

These two sources of bugs go together because people commonly mix reference-counted pointers and raw pointers precisely to avoid cycles. If two objects need to point to each other, a common strategy is to choose one of those links as owning and the other as nonowning; the owning link can be represented as a *shared_ptr*, and the nonowning link as something else. This is a perfectly valid technique, especially if you resist the temptation to use a raw pointer for that "something else." The TR1 smart pointer library provides a better alternative—*weak_ptr*. A *weak_ptr* points to an object that's already being managed by a shared_ptr; it doesn't prevent that object from being destroyed when the last *shared_ptr* goes out of scope, but, unlike a raw pointer, it also can't dangle and cause a crash, and again unlike a raw pointer, it can safely be converted into a *shared_ptr* that shares ownership with whatever other *shared_ptr*s already exist. Listing Three is an example where it makes sense to combine *shared_ptr* and *weak_ptr*.

TR1 includes other primitives as well as smart pointers, including components that make it easier to use functions and function objects. One of the most useful is the new *function* wrapper class.

Suppose you want to write something that takes two arguments of type *A* and type *B* and returns a result of type *C*. C++ gives you lots of choices for how to express that operation! You might write an ordinary function:

```
C f(A a, B b) { ... }
```

Or, if *C* is your own class, you might express this as a member function:

```
class A {
  ...
  C g(B b);
};
```

Or, as the STL does with such classes as *std::plus<T>,* you might choose to write this operation as a function object:

```
struct h {
  C operator()(A, B) const {...}
};
```

There are syntactic and semantic differences between these options, and member functions, in particular, are invoked in a different way. You can encapsulate the syntactic differences using the Standard's *mem_fun* adaptor (or more conveniently, the TR's new *mem_fn* adaptor or *bind* adaptor), but there's one thing this won't help with— the types. We have three different ways of performing the same fundamental operation, *A×B->C*, and as far as the language is concerned they all have different types. It would be useful to have some single type that could represent all of these different versions.

That's what the TR1 *function* wrapper does. The type *function<C(A,B)>* represents any kind of function that takes *A* and *B* and returns *C*, whether it's an ordinary function, member function, or function object. Again, implementing *function* is quite difficult but all of that complexity is hidden. From the user's point of view, it just does what you expect: You can instantiate the *function* template with any reasonable types (as usual there's some limit on the number of parameters, but the limit should be large), you can assign anything to it that makes sense, and you invoke it using the ordinary function call syntax.

Use *function<void(my_class)>*, for example, to hold and invoke three different kinds of functions, all of which take a *my_class* argument and return nothing; see Listing Four. Putting these *function<void(my_class)>* objects into a *vector* may seem like an unnecessary complication. I did it to give a hint about why this is useful. In one word— callbacks. You now have a uniform mechanism that higher level libraries can use to hold the callbacks they're passed by their clients. I expect that in the future we will see this mechanism in the interfaces of many new libraries, especially ones designed for dynamism and loose coupling.

Application Support: Regular Expressions

Low-level components that make it easier to write other libraries are important, but so are library components that directly solve programmers' problems. Most programs have to work with text, and one of the classic techniques for text processing is pattern matching by regular expressions. Regular expressions are used in compilers, word processors, and any program that ever has to read a configuration file. Good support for regular expressions is one of the reasons that it's so easy to write simple "throwaway" scripts in Perl. Conversely, the lack of support for regular expressions is one of C++'s greatest weaknesses. Fortunately, as of TR1, we now have that support.

TR1 regular expressions have many features and options but the basic model is quite simple; it should seem familiar if you've used regular expressions in languages like Python or Java. First, you create a *tr1::regex* object that represents the pattern you'd like to match, using the standard syntax from ECMA-262 (that is, the same syntax that JavaScript uses). Next, you use one of the regular expression algorithms (*regex_match, regex_search*, or *regex_replace*) to match that pattern against a string. The difference between "match" and "search" is that *regex_match* tests whether the string you're trying to match is described by the regex pattern, while *regex_search* tests whether the string contains something that matches the pattern as a substring. Both *regex_match* and *regex_search* return *bool*, to tell you whether the match succeeded. You can also, optionally, pass in a *match_results* object to get more details.

Actually, you probably won't use *match_results* explicitly. The TR1 regular expression library, like the standard IOStream library, is templatized, but most of the time you can ignore that feature. You probably use *istream*, and not *basic_istream*. Similarly, you will probably use *regex*, which is an abbreviation for *basic_regex<char>,* instead of using *basic_regex* directly. In the case of *match_results*, you will probably use one of two specializations: *smatch* if you're searching through a *string*, and *cmatch* if you're searching through an array of *char*.

Listing Five shows how you might use TR1 regular expressions to write the core of the UNIX grep utility. With *do_grep*, you're only concerned with whether you have a match, not with any substructure. But one of the other main uses of regular expressions is to decompose compound strings into individual fields, as in

Listing Six(a). In Listing Six(b) we take this further using regular expressions to convert between American and European customs for writing dates.

When you use *regex_search* it only shows you the first place where there's a match, even if there may be more matches later on in the string. What if you want to find all of the matches? One answer would be to do a search, find the first match, examine the *match_results* to find the end, search through the rest of the string, and so on. But that's harder than it needs to be. This is a case of iteration, so naturally you can just use an iterator. To collect all matches into a *vector*:

```
const string str =
    "a few words on regular expressions";
const regex pat("[a-zA-Z]+");

sregex_token_iterator first(str.begin(),
                            str.end(), pat);
sregex_token_iterator last;

vector<string> words(first, last);
```

The Future

Clearly, the entire technical report is too much to cover in a single article. I mentioned *shared_ptr* and *function*, but I only alluded to *reference_wrapper*, *result_of*, *mem_fn*, and *bind*. I mentioned *tuple* and the unordered associative containers, but I left out the other new STL container in TR1, the fixed-size STL container *array<T, N>*. I entirely left out type traits, because it's mostly useful to library writers (it's very exciting if you do happen to be a library writer or if you do template metaprogramming!), and I left out the extensive work on random-number generation and on mathematical special functions. Either you care deeply about Bessel functions, hypergeometric functions, and the Riemann ζ function or you don't; if you do, now you know where to find them. And finally I left out the section on C99 compatibility. That's for essentially the opposite reason as the others—it's useful, it's important, and it just works. C99 functions in C++ should work just the way you would expect.

At this writing, I'm not aware of any complete implementations of the TR1 libraries. Still, there is work being done:

- Metrowerks CodeWarrior 9.0 ships with a partial implementation of TR1, including such classes as *function*, *shared_ptr*, and *tuple*.
- Many parts of TR1, including the smart pointers, regular expressions, and random number generators, were originally Boost libraries (http://www.boost.org/). Boost releases are available, and free, for all popular compilers and platforms.
- Dinkumware is in the process of implementing the entire technical report. It is the only company I know of that's currently working on the TR1 special functions, like *cyl_bessel_j* and *riemann_zeta*, with the goal of achieving accuracy comparable to today's best implementations of functions like *sin* and *cos*.

> "With the Standards committee nearly done, it's time to think about the next round of library extensions"

- The GNU libstdc++ project, which writes the C++ library that ships with GCC, is actively working on implementing TR1. The next release of GCC, GCC 4.0, will ship with a partial implementation of TR1—exactly how partial is hard to say, since TR1 components are being added to libstdc++ on a daily basis.

TR1 is real, not vaporware. All of the code samples in this article are real; I compiled them. (Well, except for the "…" parts.) The GNU libstdc++ implementation is still experimental and incomplete, but already complete enough that I was able to use it to test the examples of unordered associative containers, *tuple, functional,* and smart pointers. Because a GNU implementation of TR1 regular expressions doesn't yet exist, I used Boost.Regex. All I had to do was change the header and namespace names.

With the Standards committee nearly done with TR1 and implementation work underway, it's time to think about the next round of library extensions. What can we expect to see? It's too early to say. The Standards committee hasn't started discussing proposals for TR2 yet. I'm maintaining a list of some of the extensions people have asked for (http://lafstern.org/matt/wishlist.html) but there's a long way between a wish and a fully fleshed out proposal.

My own wish is better support for common practical tasks: parsing HTML and XML, manipulating GIF and JPEG images, reading directories on file systems, using HTTP. I'd like simple tasks, like trimming whitespace from a string or converting it to uppercase, to be simple. We've done an excellent job of creating general infrastructure to help library writers; now it's time to use some of that power to improve the experience of day-to-day development.

What's important to remember, though, is that Standards committees standardize; they don't invent. The only things that have a chance of making it into TR2 will be the ones that individuals feel strongly enough about to do real work on. Perhaps you'll be inspired by some of the entries on the "wish list," or by my suggestions, or by libraries from Boost or some other organization. As I wrote a few years ago, when work on TR1 had just begun (see "And Now for Something Completely Different," *C/C++ Users Journal,* January 2002), a library extension proposal should explain why this particular problem area is important, what the solution should look like, how your work relates to previous work in the same area, and how your work affects the rest of the library. It's a nontrivial task, but it's easier now than it was then: One thing we have now, that we didn't before, is examples of what library extension proposals can look like. The proposals that were accepted into TR1 are collected at http://open-std.org/jtc1/sc22/wg21/docs/library_technical_report.html, and they can serve as models for TR2 proposals.

Now that the first Technical Report on C++ Library Extensions has essentially been completed, it's time to start thinking about the next round of library extensions. What comes next is partly up to you!

References

Austern, Matthew H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library,* Addison-Wesley, 1998.

Josuttis, Nicolai. *The C++ Standard Library: A Tutorial and Reference,* Addison-Wesley, 1999.

Langer, Angelika and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference,* Addison-Wesley, 2000.

Lischner, Ray. *STL Pocket Reference,* O'Reilly & Associates, 2003.

Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library,* Addison-Wesley, 2001.

Musser, David R., Gilmer Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library,* Second Edition, Addison-Wesley, 2001.

Plauger, P.J., Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*, Prentice Hall, 2000.

DDJ

(Listings begin on page 72.)

Listing One

```
#include <tr1/unordered_map>
#include <iostream>
#include <string>

int main()
{
  using namespace std;
  using namespace std::tr1;
  typedef unordered_map<string, unsigned long> Map;
  Map colors;

  colors["black"] = 0xff0000ul;
  colors["red"] = 0xff0000ul;
  colors["green"] = 0x00ff00ul;
  colors["blue"] = 0x0000ful;
  colors["white"] = 0xffffful;

  for (Map::iterator i = colors.begin();
       i != colors.end();
       ++i)
  cout << i->first << " -> " << i->second << endl;
}
```

Listing Two

```
#include <iostream>
#include <tr1/memory>

using namespace std;
using namespace std::tr1;

struct A {
  A() { cout << "Create" << endl; }
  A(const A&) { cout << "Copy" << endl; }
  ~A() { cout << "Destroy" << endl; }
};

int main() {
  shared_ptr<A> p1(new A);
  shared_ptr<A> p2 = p1;
  assert (p1 != NULL && p2 != NULL && p1 == p2);
}
```

Listing Three

```
class my_node {
public:
  ...
private:
  weak_ptr<my_node> parent;
  shared_ptr<my_node> left_child;
  shared_ptr<my_node> right_child;
};
```

Listing Four

```
#include <tr1/functional>
#include <vector>
#include <iostream>

using namespace std;
using namespace std::tr1;

struct my_class
{
  void f() { cout << "my_class::f()" << endl; }
};

void g(my_class) {
  cout << "g(my_class)" << endl;
}

struct h {
  void operator()(my_class) const {
    cout << "h::operator()(my_class)" << endl;
  }
};

int main()
{
  typedef function<void(my_class)> F;
  vector<F> ops;
  ops.push_back(&my_class::f);
  ops.push_back(&g);
  ops.push_back(h());

  my_class tmp;
  for (vector<F>::iterator i = ops.begin();
       i != ops.end();
       ++i)
    (*i)(tmp);
}
```

Listing Five

```
#include <regex>
#include <string>
#include <iostream>

using namespace std;
using namespace std::tr1;

bool
do_grep(const string& exp, istream& in, ostream& out)
{
  regex r(exp);
  bool found_any = false;
  string line;

  while (getline(in, line))
    if (regex_search(line, r)) {
      found_any = true;
      out << line;
    }

  return found_any;
}
```

Listing Six

(a)

```
const string datestring = "10/31/2004";
const regex r("(\\d+)/(\\d+)/(\\d+)");
smatch fields;
if (!regex_match(datestring, fields, r))
  throw runtime_error("not a valid date");

const string month = fields[1];
const string day = fields[2];
const string year = fields[3];
```

(b)

```
const string date = "10/31/2004";
const regex r("(\\d+)/(\\d+)/(\\d+)");
const string date2 = regex_replace(date, r, "$2/$1/$3");
```

DDJ

Measuring the Benefits of Software Reuse

Examining three different approaches to software reuse

LIOR AMAR AND JAN COFFEY

Software reuse has long been on the radar of many companies because of its potential to deliver quantum leaps in production efficiencies. In fact, basic, or ad hoc software reuse already exists within most organizations. This reuse of documents, coding styles, components, models, patterns, knowledge items, and source code is rarely discussed because it usually starts and ends as an informal grass roots effort, with management having little understanding of how it started, why it persists, and how they might proactively extract larger benefits from it.

With an understanding that some form of reuse very likely already exists within most, if not all, software development organizations, the questions emerge, "how can we measure the level of reuse that already exists?", "what can be done to increase reuse benefits?", and "how can we track our progress along the way?".

Finding a Unified Unit of Measure

The first step to being able to measure how instances of software reuse are impacting operations is to define a base unit of measure that you can use across all instances.

The primary issue in finding such a unified unit of measure lies in the fact that reuse is not limited to the source or machine code. In fact, all of the assets associated with software development are possible targets for reuse (components, source code, documents, models, web services, and the like). As a result, artifact-specific measures such as lines of code, pages of documentation, or classes in a diagram are simply not generic enough to be useful, and for the most part do not readily translate into real corporate costs. We suggest using hours as the base unit of measure. Work hours translate directly into costs for a software organization, are easily measurable, and can be universally applied across all artifacts.

Some have used "average developer hours" as the base unit of measure with average developer hours defined as the number of productive hours that an average developer typically spends directly on software development (15 hours per week, for example). Because the organization pays for software developers, regardless of whether they work on a task directly or indirectly related to the software project, we propose sticking to easily measurable worked hours as the base unit of measure since it is less subjective.

Another reason cited for using "average developer hours" is that there are cases of certain developers within the organization who are much more (or less) productive than the "average" developer. Presumably, however, developers who are extra productive will be recognized as such, and this trait will be reflected in their salary. Salary, which is a market-determined metric, is therefore likely the best and only unbiased measure that we can use to compensate for productivity differences between developers. As long as we use salaried rates for each resource as opposed to some average for the group, we should be able to implicitly keep track of these differences in productivity as we measure dollar cost savings.

Measuring Ad Hoc Software Reuse

Because there are no set-up or other costs associated with ad hoc reuse, the only costs to the enterprise relate to the time spent searching for and analyzing whether a particular reuse candidate can in fact help accelerate the development of a current task. If the search yields a positive result, there are also subsequent costs associated with modifying/integrating the reusable item into the current project. The risks associated with ad hoc reuse initiatives relate to the time spent to determine whether reuse candidates exist because this time is nonrecoverable and is added to the total development time in the event that no reuse item is located.

Over multiple search and reuse iterations, the combined time spent searching, understanding, and integrating the found content into the current project must be less than the time to develop all of the integrated content from scratch for the reuse efforts to be judged as successful.

> "Some form of software reuse exists within most organizations"

In mathematical terms, this is written as follows, where the expression on the left signifies the total time to develop the content over all reuse or attempted reuse iterations, and the expression on the right indicates the actual or expected time required to build all of the combined content from scratch:

$$(TLR+U)*N + i*SR*MOD + i*(1-SR)*BUILD < i*BUILD$$

In this case, *TLR* = Time to locate each potentially reusable item; *U* = Time to understand suitability of each potentially reusable item for current task; *N* = Number of items that were examined, including each of the items that finally get reused (if any); *i* = number of attempted instances of reuse; *SR* = Search hit rate. Percentage of *i* that yielded a positive search result (for instance, the user discovered a suitable reuse candidate that gets incorporated into the project); *MOD* = Time to integrate/modify the reused item for current purposes; and *BUILD* = Time to build an element from scratch. This is the actual or

Lior is chief technology officer at OSTnet and can be contacted at http://www.ostnet.com/. Jan is a software engineer at Inovant and can be contacted at http://www.visa.com/.

estimated time spent building the software. To calculate expected time to project completion, developers can use any estimation methods currently in place internally (project plans, function point analyses, black magic voodoo).

Similarly, by taking the percentage difference between the no reuse and ad hoc reuse scenarios (for instance, *(no reuse-ad hoc reuse)/no reuse*100*), you can arrive at the percentage of savings generated by ad hoc reuse in the enterprise. After simplifying, this equation looks as follows:

$$\% \text{ Savings} = [SR - (TLR+U)*(N/i)/BUILD - SR*(MOD/BUILD)]*100$$

In this instance, *(TLR+U)*N/i* is the average time spent searching for a reusable item before an item is found or the user decides to build from scratch. This number is typically less than five minutes. If the average size of the item that you are looking to build is over eight hours (a reasonable assumption), then this term is negligible compared to *BUILD,* and the ratio of the two terms is essentially 0.

MOD/BUILD is the relative cost of integrating an element versus building it from scratch. This value has been determined over numerous empirical studies to be in the range of 0.08 for black box component reuse to 0.85 for a small snippet of code.

We'll use an average search hit rate *SR* of 20 percent (for example, a user finds a useful item one out of every five times that he actually tries to locate something) and 0.75 for an average *MOD/BUILD* value. The *MOD/BUILD* value is on the high end of its normal range since the granularity of the things being used in an ad hoc reuse initiative is typically small, as are the incremental benefits achieved. This is a fair assumption because the reuse initiative is not being managed and the developers' source for the content being reused is not optimized (that is, the content is taken from the Internet, friends, and other unmanaged sources).

Plugging the aforementioned assumptions into the equation, we find that ad hoc reuse generates savings equal to 5 percent of development costs. Although it appears small on a percentage basis, this number can actually be quite large in dollar terms given the high total cost of the development.

For example, if a company's total IT salaries are $5 million, the 5 percent increase in productivity would equate to $250,000 in annual savings.

Evolutionary Software Reuse

Regardless of the process or processes used to develop software within an organization, there are easy to implement improvements that can be initiated to enhance the returns currently being realized with ad hoc reuse. Although the tasks and the ways of measuring results will not change from one process to the next, the artifacts to be reused and the point at which the reuse-related tasks intervene in the process will vary. By way of example, companies following an RUP process will typically reuse such things as use cases, subsystems, components, source code, and documents, and these will be accessed at various points during the elaboration, construction, and transition phases.

Without significantly altering their core development process, companies can begin to benefit to a greater degree by actively managing their existing software assets. In an "evolutionary reuse" practice, users are encouraged to identify all potentially reusable items and make them available to others in the organization, without investing any time up-front to make them "reusable." During each instance of reuse, the individual reusing the asset is encouraged to refactor the reusable artifacts and republish the upgraded asset, thereby evolving it towards black box reuse.

By following this reuse methodology, no initial investment is required to generalize the asset in anticipation of reuse that may not ever occur. Each asset is only extended to the extent needed to accommodate the current requirements, thus there are no sunk costs on assets that were created for reuse but never reused.

To implement a more structured evolutionary reuse effort, companies need to:

- Provide better access to their own internal software content.
- Promote the development of well-factored software (a process that is already quite familiar to most software developers).
- Measure results and gradually refine the reused content to ensure growing incremental benefits with each new instance of reuse.

Looking at how we model and measure evolutionary software reuse, we first need to identify all incremental costs and benefits that are not present in an ad hoc initiative. These are:

- Users who locate a reusable asset will typically need to refactor the asset for current purposes. Most of this effort is captured in *MOD*, but there may occasionally be additional effort involved with restructuring the asset to ensure that it remains well-factored. Since this effort is only necessary when something is to be reused, the total incremental cost is *i*SR*FACT,* where *FACT* is the average incremental time to refactor assets for entry into the asset repository.
- In addition to ensuring that the reusable artifacts are well factored, there are additional costs associated with creating assets from your reusable artifacts (for instance, attaching metadata to make the artifacts easier to find, understand, and reuse) and managing a repository of assets, although selecting the right repository tool for your organization can minimize these costs. These costs are accounted for as *REP* for each new asset.

Inserting these terms into the ad hoc reuse equation and taking the percentage of savings, we get (after simplifying):

$$\% \text{ Savings} = [SR - (TLR+U)*(N/i)/BUILD - SR*((MOD+FACT)/BUILD) - REP/BUILD]*100$$

As before, *(TLR+U)*(N/i)/BUILD* is approximately 0 and can be ignored. Interestingly, the term *(MOD+FACT)/BUILD* in the evolutionary reuse scenario continues to vary between 0.08 and 0.85 and, as an average, actually is smaller than *MOD/BUILD* in an ad hoc scenario. By way of example, in an ad hoc reuse scenario, if two developers reuse the same artifact on separate occasions, their efforts will likely be duplicated because the improvements made by the first developer reusing the artifact will likely not be available to the second developer (unless they know of each other's work). If one spends 20 hours modifying the artifact for reuse, the other will also likely spend a similar amount of time, resulting in a combined *MOD* of 40 hours.

In an evolutionary reuse scenario, the first developer will likely spend a few more hours modifying and refactoring the artifact to make sure that its interfaces are clean and easily consumable. Because the first developer publishes this asset after he is done, the second developer will reuse the improved asset, thus requiring only a fraction of the time to understand, modify, and refactor it (eight hours, for instance). So if the first developer spent 22 hours modifying and refactoring the artifacts, the total of *MOD+FACT* over the two reuse instances under the evolutionary reuse scenario will be only 30 hours. With over hundreds of reuse instances, it is easy to see how the average of *MOD+FACT* will continue to trend lower as the repository of software assets grows and matures. At the limit, when an asset in the repository is black boxed, *(MOD+FACT)/BUILD* will equal 0.08 because it will no longer be necessary to refactor the asset (*FACT*=0).

The term *REP/BUILD* in the equation relates primarily to the time required to publish assets as they are located. This time will vary depending on the workflow process used to publish assets and on the amount of metadata that the organization

determines is necessary to accurately describe the asset. In general, this time is very small and its costs are more than offset by the reduction in the time others spend trying to understand what an artifact does when it is located.

By following an evolutionary reuse practice, the company very quickly has at its disposal a rich asset repository filled with reusable company content that:

- Is exclusively focused on its particular domain of operation.
- Has been tested and approved for use within the company.

As a result, developers looking to reuse will quickly be able to determine whether useful reusable artifacts exist and will also be able to locate more content, with greater precision, thus increasing the search hit rate. While we will use an increased search hit rate of 40 percent in the aforementioned equation, it should be noted that the search hit rate will continue to increase as the repository grows and more content becomes available for reuse.

We will use 0.5 for an average *(MOD+FACT)/BUILD* value, which is high since the most popular assets will be reused multiple times, resulting in many cases of black box reuse (0.08) and driving down the average. Plugging in the stated numbers, we find that the evolutionary reuse scenario generates very respectable savings of 20 percent. This will amount to a dollar savings of $1 million using salaries of $5 million, as above. Interestingly, this value can be extracted without a material initial investment in time and effort to get started.

Systematic Software Reuse

When people refer to software reuse without qualifying further, they are typically speaking about traditional "systematic software reuse." Systematic software reuse is a highly structured practice that involves architects and developers identifying potentially reusable components in a project or family of projects in advance of their development.

Systematic software reuse efforts include "standards police," review committees, and/or special "tools teams" responsible for specifically developing reusable assets. Because it is believed that future modifications can be foreseen, developers practicing Systematic software reuse build in abstractions to cover any number of possible mutations and implement "hooks" for future iterations.

The end goal of all of this up-front effort is to reduce the time required to integrate the reusable component into a new project by enabling black-box software reuse to the largest extent possible (for instance, *MOD*=0.08). However, overabstracting components ahead of time can make code harder for others to read and understand and is an inadvertent problem associated with this practice.

While the leverage associated with systematic software reuse is very large because each additional instance of reuse provides enormous benefits, the added up-front costs dramatically increase the risks associated with its implementation.

To properly measure the impact that systematic software reuse can have on a development environment, we begin with the ad hoc reuse approach and add all additional tasks and their resulting benefits into the equation. Of particular note:

- Because reusable components are "built" to be reusable, there are costs associated with building these components over and above what it would otherwise cost to build them for a given set of software requirements. Industry accepted figures are that it typically costs anywhere between 50 percent and 150 percent extra to build a component for reuse, versus building it for a single use. We'll use *RCOM* to identify this extra effort in our equations (to be shown shortly). In the case of evolutionary reuse, this extra effort to make an asset reusable is only done at the time of consumption by the person who is looking to reuse the component, and this effort is captured in the term *(MOD+FACT)*.
- The cost of reusing a component built for reuse will be much lower than in other types of reuse with *MOD/BUILD* ranging between 0.08 and 0.2.
- Because systematic reuse components are built for reuse, there will typically only be a small number of them available for reuse. Also, the availability of these components should be fairly easy to communicate within the organization meaning that the Search hit rate will be much higher in a Systematic reuse effort, although the actual number of reuse instances *i* will be dramatically lower, especially in the early years.

For a systematic software reuse effort to be profitable, therefore, the following equation representing a systematic software reuse initiative must hold true:

$$(TLR+U)*N + i*SR*MOD + i*(1-SR)*BUILD + j*REP + j*(1+RCOM)*BUILD < i*BUILD$$

where *j* = number of reusable software components that have been built, and *RCOM* = extra time required to build a reusable software component versus building one with equivalent functionality but that is not designed to be reusable.

Taking the percentage difference between the no reuse and Systematic software reuse scenarios, we can arrive at the % Savings generated by systematic software reuse in the enterprise. After simplifying, this equation looks like:

$$\%\ Savings = [SR - (TLR+U)*(N/i)/BUILD - SR*MOD/BUILD - (j*REP)/(i*BUILD) - j*(1+RCOM)/i]*100$$

For demonstration purposes, and to simplify this equation, assume that the search hit rate *SR* approaches 1 and that *RCOM* is 50 percent, the low end of its industry accepted value. As well, we'll use a favorable *MOD/BUILD* value of 0.08 and will assume that *(TLR+U)*(N/i)/BUILD* approximates 0, as was the case in each of the aforementioned scenarios. Finally, we'll assume that the expression *(j*REP)/(i*BUILD)* is also equal to zero, which should be the case unless *j* (the number of reusable components that have been built and inserted into the catalog) is orders of magnitude greater than *i* (the number of reused elements), which should be the case in all but the most disastrous scenarios.

Plugging in the favorable values just listed for each expression and reducing the equation, we get:

$$\%\ Savings = [0.92 - 1.5*j/i]*100$$

What we can interpret from this equation is that the extra 50 percent spent to build each reusable component adds up very quickly and needs to be amortized over multiple reuse iterations for systematic software reuse to generate positive savings. In fact, if each item built is not used on an average of 1.63 projects (*i/j* 1.63, for instance), then the reuse effort will fail to generate a positive return.

Overall, systematic software reuse has the potential to generate very large savings (theoretically as high as 92 percent if one magical component were built that could be reused everywhere, which of course, is not really possible). On the negative side, systematic software reuse is highly sensitive to the ratio of *j/i*, meaning that participants in the initiative need to be highly skilled at predicting which reusable components need to get built to amortize them over the largest number of reuse instances. Failing to accurately pick the right components to build or mismanaging the Systematic software reuse initiative have the potential to very quickly generate costly negative results.

Comparison

Using the aforementioned methods for calculating the costs and benefits of each of the three reuse implementation methods covered and deriving an ROI from each, we arrive at the ROI graph in Figure 1.

Again, systematic software reuse has the potential to be highly negative if the assets that are built are not quickly

reused on multiple projects. Systematic reuse does, however, have the highest slope in the early days, meaning that it can provide a very quick ROI if properly implemented.

Evolutionary reuse starts off with low incremental benefits to the organization but quickly begins to generate increasing value as content is refactored and made available to end users. It provides a nice compromise for companies looking to enhance the benefits they are currently getting from their ad hoc reuse efforts but who are unwilling or unable to invest the time required to set up and manage a structured systematic reuse effort.

Finally, ad hoc reuse currently generates modest benefits to an organization and it will continue to do so, although these benefits grow slowly and are far from being optimized.

Conclusion

Measuring productivity and changes in productivity are important when implementing any new software tool or initiative. To that end, the overall techniques just used to determine the costs and benefits related to different reuse practices can also be applied to measure savings associated with other initiatives. It is only in comparing these different returns using standard methods and units of measure that you will be able to make informed decisions and set quantifiable milestones for your company.

As a starting point, additional work needs to be done by most companies to gain a better understanding of where development efforts are currently being focused, which tasks are the most costly, which are being duplicated, and can be altered to generate the highest incremental returns.

The returns just quantified relate directly to the savings that organizations can hope to gain through developer productivity enhancements. These savings are the minimum benefits realizable since they exclude all other costs (such as overhead) and tertiary benefits such as increased IT agility, reduced defects and maintenance costs, and the ability to deliver new products and services at an accelerated rate to establish or maintain key strategic competitive advantages. As we have seen, depending on the path chosen, establishing this advantage through reuse does not necessarily require a huge up-front investment in time and human resources.

DDJ

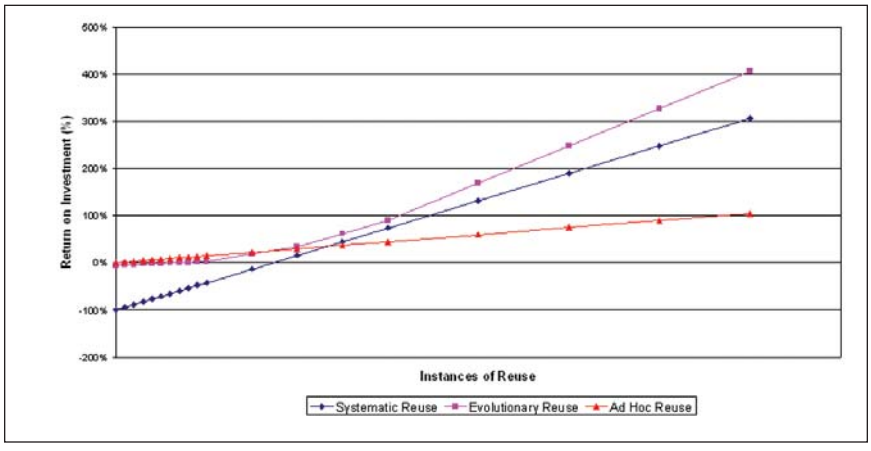


Figure 1: *Return on investment analysis.*

Loadable Modules & The Linux 2.6 Kernel

Changes to the kernel mean changes must be made elsewhere

DANIELE PAOLO SCARPAZZA

The heart of Linux is the kernel, which is in charge of scheduling tasks, managing memory, performing device I/O operations, servicing system calls, and carrying out other critical tasks. Unlike other UNIX-like operating systems that implement a pure monolithic architecture, Linux lets you dynamically load/unload portions of the kernel (modules). This lets you provide support for new devices and add system features without recompiling or rebooting the kernel, then unload them when they are not needed anymore.

The possibility of loading/unloading modules is a key feature for driver programmers because it lets you test drivers during development without rebooting the kernel at every change, thus dramatically speeding up the test-and-debug process.

Kernel 2.6 introduces significant changes with respect to kernel 2.4: New features were added, existing ones removed, and some marked as deprecated, although they're still usable but with severe limitations. Consequently, modules written for kernel 2.4 don't work anymore, or work with grave restrictions. In this article, I examine these changes.

Daniele is a Ph.D. student at Politecnico di Milano (Italy), where he currently works on source-level software energy estimation. He can be contacted at scarpaz@scarpaz.com.

A Minimal Module

Listing One is the shortest possible implementation of a module. Adhering to this template lets you write code that can operate equally as a module or statically linked into the kernel, without modifications or *#ifdef*s.

The initialization and cleanup functions can have arbitrary names, and must be registered via the *module_init()* and *module_exit()* macros. The *module_init(f)* macro declares that function *f* must be called at module insertion time if the file is compiled as a module, or otherwise at boot time. Similarly, macro *module_exit(f)* indicates that *f* must be called at module removal time (or never, if built-in). The specifier *__init* is effective only when the file is compiled in the kernel, and indicates that the initialization function can be freed after boot. On the other hand, *__exit* marks functions that are useful only for module unloading and, therefore, can be completely ignored if the file is not compiled as a module.

Compiling Modules

The 2.6 kernel's build mechanism ("kbuild") has been deeply reengineered, affecting how external kernel modules are compiled. In 2.4, module developers manually called GCC, including command-line preprocessor symbol definitions (such as *MODULE* or *__KERNEL__*), specifying *include* directories and optimization options. This approach is no longer recommended because external modules should be built as if they were part of the official kernel. Consequently, kbuild automatically defines preprocessor symbols, optimization options, and include directories. The only required thing you do is create a one-line makefile:

```
obj-m := your_module.o
```

"The 2.6 module loader implements strict version checking"

where *your_module* is the name of your module, whose source is in the file your_module.c. You then type a command line such as:

```
make -C /usr/src/linux-2.6.7 SUBDIRS=
                              'pwd' modules
```

The output provided by the build process is:

```
make -C /usr/src/linux-2.6.7 SUBDIRS=
                    /root/your_dir modules
make[1]: Entering directory
                    '/usr/src/linux-2.6.7'
  CC [M]  /root/your_dir/your_module.o
  Building modules, stage 2.
  MODPOST
  CC      /root/your_dir/your_module.mod.o
  LD [M]  /root/your_dir/your_module.ko
```

```
make[1]: Leaving directory
                    '/usr/src/linux-2.6.7'
```

In the end, a new kernel module is available in your build directory under the name of *your_module*.ko (the .ko extension distinguishes "kernel objects" from conventional objects). With a more elaborate Makefile (such as Listing Two), you can avoid typing this command line.

Module Versioning

The 2.6 module loader implements strict version checking, relying on "version magic" strings ("vermagics"), which are included both in the kernel and in each module at build time. A vermagic, which could look like "2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3," contains critical information (for example, an extended kernel version identifier, the target architecture, compilation options, and compiler version) and guarantees compatibility between the kernel and a module. The module loader compares the module's and kernel's vermagics character-for-character, and refuses to load the module if differences are detected. The strictness of this check complicates things, but was advocated after compatibility problems arose when loading modules compiled with different GCC versions with respect to the kernel.

When compiling modules for a running kernel that you may not want to recompile, when cross compiling for a deployment box that you do not want to reboot, or when preparing a module binary for a kernel provided with a given Linux distribution, your module's vermagic must exactly match your target kernel's vermagic. To do this, you must exactly duplicate the build environment during module compilation, to that present at kernel compilation time. This is done by:

1. Using the same configuration file as the kernel (since the configuration file used to compile the kernel is available in most cases under /boot, a *cp /boot/config-'uname -r' /usr/src/linux-'uname -r'/.config* command is enough in most cases).
2. Using the same kernel top-level Makefile (again, it should be available under /lib/modules/2.6.x/build; therefore, the command *cp /lib/modules/'uname -r'/build/Makefile /usr/src/linux-'uname -r'* should go).

Module Licensing

The Linux kernel is released under the GNU Public License (GPL), whose purpose is to grant users rights to copy, modify, and redistribute programs, and to ensure that those rights are preserved in derivative works:

> 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licenser to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

The practical case of a kernel module depending on a second module (a common case in Linux) is not explicitly mentioned in the GPL, yet some interpretations of its underlying philosophy postulate that a proprietary module should not depend on a GPL-licensed one, because the latter would restrict the rights granted to the user by the former. Module writers advocating this interpretation can now enforce this policy with the *EXPORT_SYMBOL_GPL()* macro in place of *EXPORT_SYMBOL()*, thus exporting symbols that can be linked only by modules specifying a GPL-compatible license.

With this in mind, all module writers are asked to declare the license under which their module is released, via the macro *MODULE_LICENSE()*. Table 1 lists the licenses and respective *indent* strings currently supported by the kernel (all *indent* strings indicate free software except for the last one). Additionally, the indication of license makes it possible for users to verify that their system is free, the free development community can ignore bug reports including proprietary modules, and vendors can do likewise based on their own policies.

When no license is specified, a proprietary license is assumed. Modules with a proprietary license cause the following warning when loading:

```
your_module: module license
                    'Proprietary' taints kernel.
```

and *force* flags must be specified to have the module properly loaded.

The macro *EXPORT_NO_SYMBOLS* is deprecated and not needed anymore because a module exporting no symbols is the norm.

indent string	Meaning
GPL	GNU Public License v2 or later.
GPL v2	GNU Public License v2.
GPL and additional rights	GNU Public License v2 rights and more.
Dual BSD/GPL	GNU Public License v2 or BSD license choice.
Dual MPL/GP	GNU Public License v2 or Mozilla license choice.
Proprietary	Nonfree products.

Table 1: indent *strings for licenses currently recognized by the kernel and their respective license names.*

OO Concept	Kernel Object Concept
object	kobject
class	ktype
generic container	kset
class of a given object	ktype pointed by field ktype in given kobject
destructor	function pointed by field release in given ktype
methods	functions pointed by fields sysfs_ops.show and sysfs_ops.store of ktype
'this' pointer	first parameter (struct kobject * kobj) for the above

Table 2: *Object-oriented/kernel object mappings.*

Parameter Passing

The old parameter passing mechanism, based on the *MODULE_PARM()* macro, is obsolete. Modules should define their parameters via a call to the macro *module_param()*, whose arguments are:

- The name of the parameter (and associated variable).
- Its type (chosen among *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *charp*, *bool*, and *invbool*, or a custom typename; for example, named xxx, for which helper functions *param_get_xxx()* and *param_set_xxx()* must be provided).
- The permissions for the associated sysfs entry—0 indicates that the attribute is not to be exposed via sysfs.

Example 1 presents two example declarations.

Use Count

Module use counts protect against the removal of a module that is still in use. Modules designed for previous kernels called *MOD_INC_USE_COUNT()* and

MOD_DEC_USE_COUNT() to manipulate their use count. Since these macros could lead to unsafe conditions, they are now deprecated. They should now be avoided, for example, by setting the owner field of the *file_operations* structure, or replaced with *try_module_get()/module_put()* calls. Alternatively, you can provide your own locking mechanism in a custom function, and set the module's *can_unload* pointer to it. The function should return *0* for "yes," and *-EBUSY* or a similar error number for "no."

If used, the deprecated *MOD_INC_USE_COUNT* macro marks the current module as unsafe, thus making it impossible to unload (unless enabling the forced unload kernel option and using *rmmod -force*).

The 2.6 Device Model and /sys Filesystem

Kernel 2.6 introduces an "integrated device model"— a hierarchical representation of the system structure, originally intended to simplify power-management tasks. This model is exposed to user space through sysfs, a virtual filesystem (like /proc), usually mounted at /sys. By navigating sysfs, you can determine which devices make up the system, which power state they're in, what bus they're attached to, which driver they're associated to, and so on. sysfs is now the preferred and standardized way to expose kernel-space attributes; module writers should then avoid the soon-to-be obsolete procfs.

Figure 1 (available electronically; see "Resource Center," page 3) is a typical sysfs tree. The tree is conceptually similar to the view provided by the Windows "hardware manager." The first-level entries in /sys are:

- *block*, which enumerates all the block devices, independently from the bus to which they are connected.
- *bus*, which describes the structure of the system in terms of buses and connections.
- *class*, which provides device localization based on device class (the mouse, for example) apart from its physical bus connection or device numbering.
- *devices*, which enumerate all the devices composing the system.
- *firmware*, which provides a facility for the dynamic management of firmware.
- *power*, which provides the ability to control the system-wide power state.

Given the first-level classification, the same device can appear multiple times in the tree. Symbolic links are widely used to connect identical or related entities; for example, the block device *hda* is represented by a directory entry */sys/block/hda*, which contains a link named "device" pointing to */sys/devices/pci0000:00/0000:00:07.1/ide0/0.0*. The same block device also happens to be the first device connected to the IDE bus; thus, entry */sys/bus/ide/devices/0.0* points to the same location. Conversely, a link is provided pointing to the block device associated to a given device; for example, in */sys/devices/pci0000:00/0000:00:07.1/ide0/0.0*, a link named "block" points to */sys/block/hda*.

Exposing module attributes via sysfs requires a minimal understanding of

```
module_param(my_integer_parameter, int, S_IRUSR ¦ S_IWUSR );
MODULE_PARM_DESC(my_integer_parameter, "An integer parameter");
module_param(my_string_parameter, charp, S_IRUSR ¦
                                    S_IWUSR ¦ S_IRGRP ¦ S_IROTH);
MODULE_PARM_DESC(my_string_parameter, "A character string parameter");
```

Example 1: *Declaration of an integer and a string parameter.*

```
# make
make -C /usr/src/linux-2.6.5-1.358 SUBDIRS=/root/sysfs_example modules
make[1]: Entering directory '/usr/src/linux-2.6.5-1.358'
  CC [M]  /root/sysfs_example/sysfs_example.o
  Building modules, stage 2.
  MODPOST
  CC      /root/sysfs_example/sysfs_example.mod.o
  LD [M]  /root/sysfs_example/sysfs_example.ko
make[1]: Leaving directory '/usr/src/linux-2.6.5-1.358'
# insmod sysfs_example.ko
# cd /sys
# ls
block  bus  class  devices  firmware  power  sysfs_example
# cd sysfs_sample
# ls
hex  integer  string
# cat hex
0xdeadbeef
# cat integer
123
# cat string
test
# echo 0xbabeface > hex
# echo 456 > integer
# echo hello > string
# cat hex
0xbabeface
# cat integer
456
# cat string
hello
# cd ..
# rmmod sysfs_example
# ls
block  bus  class  devices  firmware  power
```

Example 2: *Interactive session in which the module sysfs_example.c is compiled, inserted, and tested. Attributes are inspected, changed, and inspected again.*

the device model and of its underlying kobjects, ktypes, and ksets concepts. Understanding those concepts is easier in an object-oriented perspective because all are C-language *structs* that implement (with debated success) a rudimentary object-oriented framework. Table 2 is a mapping between OO and kobject concepts.

Each directory in sysfs corresponds to a kobject, and the attributes of a kobject appear in it as files. Reading and writing attributes corresponds to invoking a show or a store method on a kobject, with the attribute as an argument. A kobject is a variable of type *struct kobject*. It has a name, reference count, pointer to its parent, and ktype. C++ programmers know that methods are not defined on an object basis; instead, all the objects of a given class share the same methods. The same happens here, the idea of class being represented by a ktype. Each kobject is of exactly one ktype, and methods are defined for ktypes (usually functions to show and store attributes, plus a function to dispose of the kobject when its reference count reaches zero: a destructor, in OO terms). A kset corresponds to a generic linked list, such as a Standard C++ Library generic container. It contains kobjects and can be treated as a kobject itself. Additionally, handlers can be associated to events of kobjects entering or leaving a set, thus providing a clean way to implement hot-plug operations. The cleanness of the design of such a framework is still debated.

sysfs_example.c (available electronically; see "Resource Center," page 3), a complete example of a kernel module, shows how to create and register kobject variables to expose three attributes to user space — a string and two integers, the first read and written as a decimal number and the second as a hexadecimal one. Example 2 is an example of interaction with that module.

Removed Features

Some features have been removed from 2.4. For instance, the system call table is no longer exported. The system call table (declared as *int sys_call_table[];*) is a vector containing a pointer to the routine to be invoked to carry out that call for each system call. In 2.4 kernels, this table was visible to — and, more important, writable by — any module. Any module could easily replace the implementation of any system call with a custom version, within a matter of three lines of code. Apart from possible race conditions issues (on SMP systems, a system call could be replaced while in use by an application on another processor), this implied putting an incredible amount of power — the ultimate heart of the OS — in the hands of any external module. If you're not convinced about the relevance of this danger, look into how easy it was to write and inject malicious code in the form of modules that replace *sys_call_table* entries. Implementing rootkits is possible in no more than 30 lines of code (see http://www.insecure.org/).

Concern is not only related to malicious modules, but also to proprietary modules provided in binary form only for which it is hard to tell exactly what they may do. The issue was radically eradicated in kernel 2.6: The system call table can only be modified by code built in the kernel, whose source is therefore available.

DDJ

Listing One

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
static int __init minimal_init(void)
{
  return 0;
}
static void __exit minimal_cleanup(void)
{
}
module_init(minimal_init);
module_exit(minimal_cleanup);
```

Listing Two

```
obj-m   := your_module.o
KDIR    := /usr/src/linux-$(shell uname -r)
PWD := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
install: default
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules_install
clean:
    rm -rf *.o *.ko .*.cmd *.mod.c .tmp_versions *~ core
```

DDJ

ASP.NET & Multiplatform Environments

Building blocks for .NET web apps

MARCIA GULESIAN

Running .NET web applications in the enterprise means accommodating a myriad of servers and browsers, many with distinct behaviors (see Figure 1). The traditional approach to building complex web apps for such environments is to write separate versions of your code—each meant to run correctly with an individual server and browser pair. In this article, I examine some of the challenges you face when creating a single version of .NET web apps, so that they function the same way no matter which server it's deployed to, and whichever client it's downloaded to.

Thin clients, in which data is managed and stored centrally, are being adopted for a number of reasons. For one thing, federal regulations (Sarbanes-Oxley, the Homeland Security Act, and HIPPA, among others) dictate that internal documents and communications be secured to a heretofore-unseen level, and security is easier to achieve when data is managed and stored centrally. Second, there's the high total-cost-of-ownership of the PC desktop. Additionally, networks have gotten faster, with most businesses running 100-Mpbs Fast Ethernet or 54-Mbps 802.11g Wi-Fi networks (both more than fast enough for thin-client computing). Finally, many vendors are shipping thin clients—with or without embedded Windows XP but all with web browsers—with enough resources locally that you don't waste time waiting for your system to painfully render its GUI.

At the same time, Linux is being supported by vendors such as IBM and Oracle, and .NET apps, which can run on Linux (and other flavors of UNIX), are not dependent solely on the Windows IIS application server and your web browsers—a one-to-many relationship. Enterprise-based .NET apps are now running in many-to-many server-browser pair environments, as in Figure 1. Finally, Linux is appearing on mainframes and other powerful computers that manage and store data centrally for large numbers of users.

On the server side, the standardization of C# and .NET's Common Language Runtime (CLR) lets you use open-source tools that are based on a language that is an international standard and compatible with both Microsoft and various UNIXs. This has given rise to initiatives such as Mono, an open-source development platform based on the .NET Framework that lets you build cross-platform applications (http://www.mono-project.com/). Mono's .NET implementation is based on the ECMA standards for C# and the Common Language Infrastructure (CLI). While Mono includes both developer tools and the infrastructure needed to run .NET client and server applications, I focus here on ASP.NET apps developed with Microsoft's

> "The default ASP.NET 1.1 validation controls do not provide a working client-side script model for nonMicrosoft browsers"

Marcia is chief technology officer at Gulesian Associates and author of more than 100 articles on software development. She can be contacted at marcia.gulesian@verizon.net.

Visual Studio .NET and deployed to both Microsoft's IIS (Windows) and the Apache Software Foundation's Apache HTTP server (after the addition of the Mono module).

On the client side, .NET apps downloaded to different browsers (running on a "thin" or "thick" client) exhibit different behaviors as a function of both the browser and the server from which it was downloaded. I first review how adjustments in the .NET configuration files can compensate for the problematic behavior of certain browsers when they download an app from the IIS application server (Windows). Then, I show how Mono can be used to mask the behaviors of these same browsers when downloading the same app from an Apache (Mono) server.

As Figure 1 suggests, the plethora of server-browser combinations is too large for a single article. However, I present a number of representative cases that can be used as building blocks to creating a single version of a .NET web app that functions the same way whichever server it is deployed to and whichever client downloads it.

Uplevel and Downlevel Browsers

Browsers are split into two distinct groups: uplevel and downlevel browsers. These groups define the type of native support a browser offers, and generally determine the rendering and behavior of a page downloaded from a web server.

Browsers that are considered uplevel at minimum support ECMAScript (JScript, JavaScript) 1.2; HTML 4.0; Microsoft Document Object Model (MSDOM); and Cascading style sheets (CSS).

On the other hand, downlevel browsers only support HTML 3.2 (http://aspnet.4guysfromrolla.com/demos/printPage.aspx?path=/articles/051204-1.aspx).

In practice, only modern Microsoft Internet Explorer versions fall into the uplevel category; most other browsers fall into the downlevel category.

Server controls such as dropdown lists and text boxes can behave differently for each browser type. If users have uplevel browsers, the server controls generate client-side JavaScript that manipulates the MSDOM and traps the action events directly on the client. If users have downlevel browsers, the controls generate standard HTML, requiring that browsers

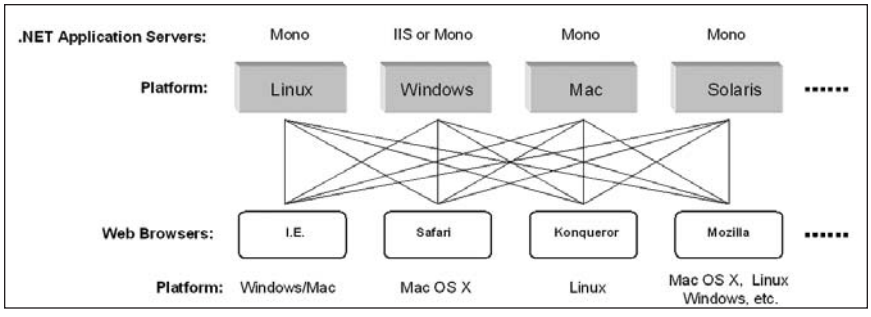


Figure1: *Server and browser pairs, some with different behaviors.*

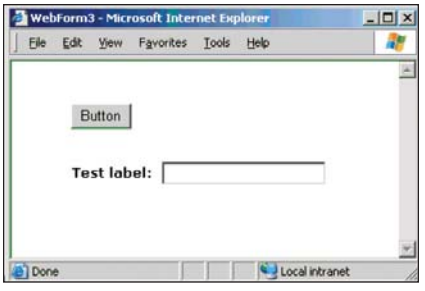


Figure 2: *WebForm rendered in IE 6.0.*

perform round-trips to the server for triggered action events.

Because different web browsers — and different versions of the same browser — have different capabilities and behaviors, web developers usually have to change their applications based on which user's browser their code detects. They use two general approaches to this problem:

- Code (typically JavaScript) is sent along with the page to be executed client-side.
- The user-agent string from the HTTP requests headers is analyzed server-side and only the appropriate HTML code is sent to the client.

Often a combination of the two is employed (see http://msdn.microsoft.com/asp.net/using/migrating/phpmig/whitepapers/compatibility.aspx?print=true and http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconwebforms-controlsbrowsercapabilities.asp).

In addition, given the existence of Rhino (Mozilla.org's JavaScript implementation in Java) and IKVM.NET (a JVM implementation for the CLR), it should be possible to run JavaScript directly under Mono (see http://chimpen.com/things/archives/001427.php).

ASP.NET's Adaptive Rendering

Figure 2 shows a web app downloaded from a Microsoft IIS application server by an Internet Explorer 5.5 or later browser running on a PC. The rendering is the same as the original design of the app in a Visual Studio .NET IDE. However, .NET controls such as single- and multiline text boxes or labels appear distorted on the page when deployed to an IIS application server and downloaded by a downlevel browser such as Safari or Konqueror; see Figures 3(b) and 4(b). That's because the HTML rendered by .NET web controls depends on the browser requesting the ASP.NET web page. And, Safari and Konqueror browsers render HTML 3.2-compliant HTML by default, in this situation. However, adding Listings One and Two to your web.config (or machine.config) file causes these browsers to render the .NET web controls of your app (or all apps running on the server) using HTML 4.0-compliant markup.

When Listings One and Two are added to the <*browserCaps*> section of the Machine.config file on the IIS server where .NET web apps are running, the web controls in all apps running on that host render without distortion in Safari (Mac), Konqueror (Linux), and Internet Explorer (PC) browsers; see Figures 2, 3(a), and 4(a). What is most interesting is that the undistorted rendering in Figures 3(a) and 4(a) is also seen in all browsers when the same application is copied to and downloaded from the Mono server without the use of Listings One and Two! To help account for changes in the browser marketplace, cyScape (http://cyscape.com/browsercaps/) publishes regular updates for the browsercaps section of your machine.config file. It is important that you keep this data current. Otherwise, pages that depend on browser detection may not operate as expected due to changes in the browser marketplace.

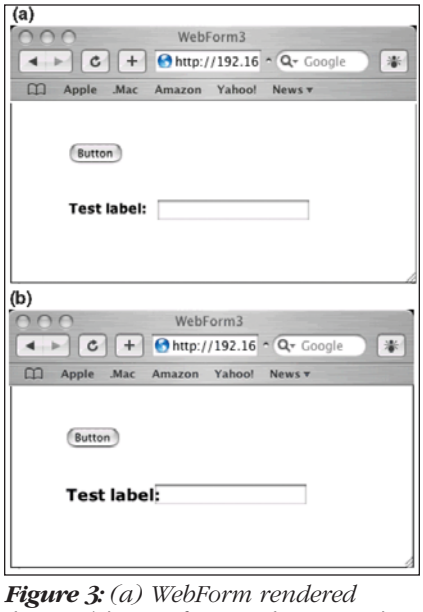


Figure 3: *(a) WebForm rendered (correctly) in Safari 1.2 (Mac OS X) with Listing One in machine.config; (b) WebForm rendered malformed in Safari 1.2 (Mac OS X) without Listing One in machine.config.*

ASP.NET Validation

The default ASP.NET 1.1 validation controls do not provide a working client-side script model for nonMicrosoft browsers, due to the fact that the proprietary *script document.all[ID]* is used in place of the standards-compliant script *document.getElementById(ID)* for referencing an element in the HTML document. If you look at the View Source in a downlevel browser and compare the code with the View Source of the IE browser, the client-side code for the validation controls is absent in the downlevel browser's View Source. Client-side support can be added, but at the cost of recreating the validation controls. Fortunately, the work has already been provided by Paul Glavich (http://weblogs.asp.net/pglavich/), so you can use his DomValidators validation controls if you need to support client-side validation with nonMicrosoft browsers.

Other solutions are provided by third-party tools such as Peter Blum's Validation Controls (http://www.peterblum.com/), which emit client-side validation code in Safari browsers. Blum's solution requires some work to install and configure before Visual Studio .NET 2003 can take advantage of these components, but they do work well. Another solution (my preference) is to hand-code custom server-side validation, or you can settle for client-side validation in IE and server-side validation in all other browsers. Or you can wait until ASP.NET 2.0 ships.

It's worth noting that this technique does have JavaScript issues. For instance, Listing Three works in IE, Safari, and Konqueror browsers, Listing Four works only in IE browsers, and Listing Five works in IE (PC) but not Safari (Mac OS X) or Konqueror. (Listing Five exploits a security bug in some browsers, IE for instance, that lets you close the current window even if it wasn't opened with client-side scripting.

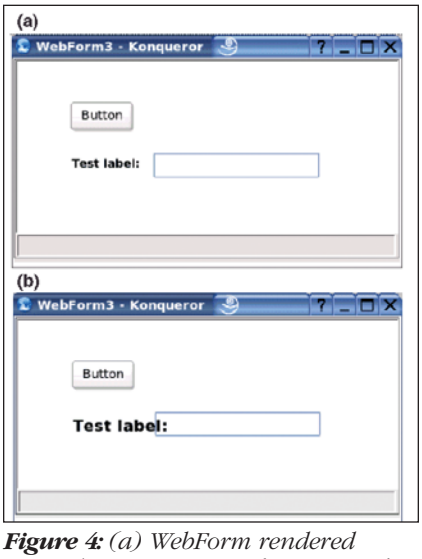


Figure 4: *(a) WebForm rendered correctly in Konqueror (SuSE Linux) with Listing Two in machine.config; (b) WebForm rendered malformed in Konqueror (SuSE Linux) without Listing Two in machine.config.*

Conclusion

With Mono and most all Linux distributions bundling Java support, it's important to include Java in any discussion that considers thin clients. In mid 2004, IBM began offering a Java-based thin software application, called "Workplace," intended for web-based applications. And, the comparative examples of JavaScript code presented here apply equally well to servlet- and JSP-based Java apps.

However, it's also important not to compare apples with oranges. At the end of 2004, you were likely to have been developing with .NET 1.1 and/or Mono 1.0 and/or JDK 1.4. In the coming months, however, you can add .NET 2.0, Mono 1.2, and JDK 1.5 to the mix.

Of course, C# and Java are playing leapfrog. C# started out with many of Java's features and some useful improvements of its own, and now Java is taking a number of C# features — attributes, *enums, foreach,* and autoboxing — and adding generics, static imports, and extendable enums. With the release of ASP.NET 2.0, Microsoft will reduce the amount of coding required for a normal web site drastically, in some cases more than 50 percent. Microsoft has also added to all the out-of-the box User controls and Validation controls, and created a new concept of Master pages, which should reduce the size of your web site. With J2EE 5.0 (previously J2EE 1.5), the Java community is likewise making it easier for less-experienced developers to create applications.

The bottom line, as suggested by Figure 1, is that .NET apps have now followed J2EE apps into the world of multiplatform deployment, which calls for a new and expanding skill set on the part of .NET developers. While both the Mono and .NET Frameworks need to be considered during the planning stage of your next ".NET" web application, this consideration needs to include your ability to work with operating systems and browsers other than just Windows Server and Internet Explorer, respectively. Failure to do so can put you at a competitive disadvantage.

References

Mark Easton and Jason King, *Cross-Platform .NET Development*. Apress, 2004.

Brian Nantz, *Open Source .NET Development*. Addison-Wesley, 2005

DDJ

Listing One

```
<!-- AppleWebKit Based Browsers (Safari...) //-->
  <case match="AppleWebKit/(?'version'(?'major'\d)
                                        (?'minor'\d+)(?'letters'\w*))">
    browser=AppleWebKit
    version=${version}
    majorversion=${major}
    minorversion=0.${minor}
    frames=true
    tables=true
    cookies=true
    javascript=true
    javaapplets=true
    ecmascriptversion=1.5
    w3cdomversion=1.0
    css1=true
    css2=true
    xml=true
    tagwriter=System.Web.UI.HtmlTextWriter
    <case match="AppleWebKit/(?'version'(?'major'\d)
                                        (?'minor'\d+)(?'letters'\w*))
                                        (\(KHTML, like Gecko\) )?
                                        (?'type'[^/\d]*)/.*$">
    type=${type}
    </case>
  </case>
```

Listing Two

```
<!-- Konqueror //-->
<case match = "Konqueror/(?'version'(?'major'\d+)
                          (?'minor'\.\d+)(?'letters'));\w*(?'platform'[^\)]*)">
browser=Konqueror
version=${version}
majorversion=${major}
minorversion=${minor}
platform=${platform}
type=Konqueror
frames=true
tables=true
cookies=true
javascript=true
javaapplets=true
ecmascriptversion=1.5
w3cdomversion=1.0
css1=true
css2=true
xml=true
tagwriter=System.Web.UI.HtmlTextWriter
</case>
```

Listing Three

```
function disableTextBox() {
var selectElement = document.getElementById('ddlWhatever');
var len = selectElement.options.length;
for (var i= 0; i < len; i++){
var bln = selectElement.options[i].selected;
var val = selectElement.options[i].value;
if (bln == true){
if (val == 'ABC'){
document.Form1.TextBox1.disabled = true;
                //  Works in I.E. (PC), Safari 1.0.2 & 1.2.2, and Konqueror
//document.Form1.TextBox1.readOnly = true;
                //  Works in I.E. (PC), Safari 1.2.2, and Konqueror
//document.getElementById("TextBox1").setAttribute("readOnly",true);
//  Works only in I.E. (PC)
}
else{
document.Form1.TextBox1.disabled = true;
            //  Works in I.E. (PC), Safari 1.0.2 & 1.2.2, and Konqueror
//document.Form1.TextBox1.readOnly = true;
            //  Works in I.E. (PC), Safari 1.2.2, and Konqueror
//document.getElementById("TextBox1").setAttribute("readOnly",true);
            //  Works only in I.E. (PC)
}
}
}
}
```

Listing Four

```
function launchWindow() {
if (document.getElementById("ddlWhatever").getAttribute("value") == 'ABC')
{
var dateWin;
dateWin = window.open("Page1.aspx",'');
dateWin.focus();
}
else
{
var dateWin;
dateWin = window.open("Page2.aspx",'');
dateWin.focus();
}
}
```

Listing Five

```
window.opener=self;
window.close();
```

DDJ

Hardware-Assisted Breakpoints

Accessing XScale debug registers from C/C++

DMITRI LEMAN

When debugging applications and drivers on Pocket PC PDAs, I often miss being able to use hardware-assisted breakpoints. When such breakpoints are enabled, the CPU runs at normal speed, stopping only when data at a given address is accessed or modified (such data breakpoints are often called "watchpoints"). While these breakpoints are not for everyday use, they can dramatically speed up the debugging of corrupted data or the exploration of unfamiliar code. Unfortunately, the Microsoft eMbedded Visual C++ (EVC) debugger does not support hardware-assisted breakpoints (at least at this writing). Granted, EVC does provide a dialog for setting data breakpoints, but it appears to implement this feature by running the program step-by-step and checking the data—a process too slow to be useful. This is unfortunate because you can substitute other debugger features, such as regular code breakpoints, by inserting trace statements or message boxes in the program itself. Still, there's no substitute for hardware breakpoints.

I recently needed software-controlled data breakpoints when debugging a large and unfamiliar code base. I noticed that a local variable in certain functions sometimes changed its value erroneously. When I tried to step through the function in a debugger or insert a breakpoint within the function, the program timing was disrupted enough to hide the bug. Consequently, I decided to write a C++ class that would set the data breakpoint in its constructor, and remove it in the destructor with minimal overhead. Then I would only need to instantiate the class in the function and run the program. After I wrote the class and ran the program for a few minutes, the data breakpoint was triggered and the debugger displayed the exact line that modified the variable in question. While I originally implemented this class for a Pentium-based Windows NT using the *SetThreadContext* API, I recently implemented it on a PocketPC PDA based on the Intel XScale architecture.

In this article, I explain how to access debug registers on XScale-based CPUs from C/C++ applications. Using the code I present here (available electronically; see "Resource Center" page 3), you can easily set breakpoints on data reading and/or writing and catch exceptions generated by these breakpoints. Also, I show how to use another feature of XScale—the trace buffer—which lets you collect program execution history. I've tested the code on several off-the-shelf XScale-based PocketPC PDAs with the Windows Mobile 2003 and Windows Mobile 2003 Second Edition operating systems. To find out if your PDA is running an XScale, open the About Control Panel applet. XScale CPUs have names starting with PXA; for example "PXA270." The code I present here won't work with ARM-compatible CPUs from manufacturers that do not support XScale debug extensions.

> "XScale-based CPUs normally run with debug functionality disabled"

Intel's XScale Architecture

Intel's XScale architecture is a successor to StrongARM, which was originally designed by Digital Equipment Corporation. At this writing, most models of Windows Mobile/PocketPC 2003 and 2002 PDAs run XScale-based CPUs, while some older PocketPC 2002 PDAs used StrongARM-based processors. All these processors are based on the ARM architecture designed

Dmitri is a consultant in Silicon Valley specializing in embedded system integration, driver, and application development. He can be reached at dmitril@forwardlab.com.

by ARM Limited. StrongARM was based on ARMv4 (ARM Version 4) and XScale on ARMv5TE. Compared to StrongARM, XScale has several extensions, such as support for the Thumb 16-bit instruction set (in addition to the 32-bit ARM instruction set), DSP extensions and debug extensions. For user mode applications, XScale maintains compatibility with StrongARM. To learn more about ARM architecture, registers, instructions, and addressing modes, see *The ARM Architecture Reference Manual,* Second Edition, edited by David Seal (Addison-Wesley, 2000). For a quick reference to XScale-supported instructions, see *XScale Microarchitecture Assembly Language Quick Reference Card* (http://www.intel.com/design/iio/swsup/11139.htm). XScale-specific features, such as memory management, cache, configuration registers, performance monitoring, and debug extensions are documented in Intel's *XScale Core Developer's Manual* (http://www.intel.com/design/intelxscale/273473.htm).

Using XScale Debug Extensions

Normally, XScale-based CPUs run with debug functionality disabled, but they may be configured to execute in one of two debug modes—Halt and Monitor. Halt mode can only be used with an external debugger connected to an XScale CPU through JTAG interface. Since off-the-shelf PDAs are unlikely to have JTAG connectors, I focus here on Monitor debug mode, which can be used by software running on the CPU itself without any external hardware or software. Useful features in this mode include instruction breakpoints, data breakpoints, software breakpoints, and a trace buffer. Except for instruction breakpoints (which are generated by a special instruction inserted into the program), these features can be enabled and configured using debug registers.

Intel provides the XDB Browser, a powerful visual debugging tool (included with the Intel C++ compiler), which gives you full control of XScale CPU internals, including debug extensions. Unfortunately, this tool requires special debug code that's built into the Board Support Package (BSP), which was unavailable on most PDAs at the time of writing.

The debug registers in Table 1 belong to coprocessors 14 (CP14) and 15 (CP15). Coprocessors are modules inside the CPU, which extend the core ARM architecture. The coprocessor registers are accessed using special commands. In the code accompanying this article, I use the commands *MRC* and *MCR* with the syntax: *MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>* to move from coprocessor to ARM register, and *MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>* to move from the ARM register to coprocessor.

- *{cond}* is an optional condition (in ARM, most instructions can be marked with a condition to specify whether the instruction should be executed or skipped depending on processor flags).
- *p<cpnum>* is either the *p14* or *p15* coprocessor name.
- *Rd* is a general-purpose ARM register.
- *CRn* and *CRm* identify the coprocessor register.
- *<op1>* and *<op2>* are opcodes (and are always 0 when working with debug registers).

Register Name	Purpose	CRn	CRm
(a)			
TX Register (TX)	Communication with JTAG debugger.	8	0
RX Register (RX)	Communication with JTAG debugger.	9	0
Debug Control & Status Register (DCSR)	Flags for enabling debug mode.	10	0
Trace Buffer Register (TBREG)	Reading from trace buffer.	11	0
Checkpoint Register 0 (CHKPTO)	Reference address for use with trace buffer.	12	0
Checkpoint Register 1 (CHKPT1)	Reference address for use with trace buffer.	13	0
TXRX Control Register (TXRXCTRL)	Communication with JTAG debugger.	14	0
(b)			
Process ID (PID)	Remapping current process addresses to slot 0.	13	0
Instruction breakpoint register 0 (IBCR0)	Address of instruction breakpoint 0.	14	8
Instruction breakpoint register 1 (IBCR1)	Address of instruction breakpoint 1.	14	9
Data breakpoint register 0 (DBR0)	Address of data breakpoint 0.	14	0
Data breakpoint register 1 (DBR1)	Address of data breakpoint 1.	14	3
Data Breakpoint Control Register (DBCON)	Configures data breakpoints.	14	4

Table 1: *XScale debug registers and PID: (a) CP14 registers; (b) CP15 registers.*

For example:

```
MCR p15, 0, R0, c14, c0, 0 ; write R0 to DBR0
MRC p14, 0, R1, c10, c0, 0 ; read DBGCSR to R1
```

Software access to debug registers can be done from a privileged mode only; user-mode access generates exceptions. Fortunately, it appears that PocketPCs always run applications in Kernel mode. Windows Mobile-based Smartphones, on the other hand, run applications in user mode. Trusted applications (which are signed with a trusted certificate) can switch to Kernel mode using the *SetKMode* API. Because I don't have an XScale-based Smartphone, I focus here on the PocketPC.

I implemented the debug register access code in assembly language; see AccessCoproc.s (available electronically), which contains several short routines: *SetDebugControlAndStatus, SetDataBreakPoint, SetCodeBreakPoint, ReadTraceBuffer, GetProgramStatusRegister,* and *ReadPID*. The file Breakpoint.file contains declaration of these functions and related constants to let you call the functions from C or C++.

To enable debug functionality, bit 31 (Global Enable) should be set in Debug Control and Status Register (DCSR). Bits 0 and 1 in this register are used to enable trace buffer. See the *SetDebugControlAndStatus* implementation in Listing One. Applications should call *DWORD dwOrigDCSR = SetDebugControlAndStatus(DEF_GlobalDebugEnabled, DEF_GlobalDebugEnabled)* before setting any breakpoints, then save the result. Before exiting, applications should call *SetDebugControlAndStatus(dwOrigDCSR, -1)* to restore the DCSR to the original value; see the *WinMain* function in BreakPointSamples.cpp (available electronically).

There are two data breakpoint registers: *DBR0* and *DBR1*. There is also Data Breakpoint Control Register (DBCON), which lets you configure hardware breakpoints on up to two separate addresses or a single breakpoint on a range of addresses. The breakpoints can be configured for load only, store only, or any (load or store) access type. To set a breakpoint on a range, *DBR0* should be set to the address and *DBR1* to a mask. The breakpoint is triggered if a program accesses data at the address that matches the value in *DBR0* while ignoring bits, which are set in the mask. I implemented an assembly routine *SetDataBreakPoint* (in AccessCoproc.s),

which assigns all three of these registers. *Enum XScaleDataBreakpointFlags* (in Breakpoint.h) defines configuration values for *DBCON*, which can be passed as the third argument to the function. For a convenient way to set breakpoints on local variables, use *DataBreakPoint*. The functions *TestWriteBreakpoint, TestReadBreakpoint,* and *TestRangeBreakpoint* in BreakPointSamples.cpp show an example. When a data breakpoint is hit, it generates a data abort exception.

The Instruction Breakpoint Address and Control Registers *IBCR0* and *IBCR1* can be used to set breakpoints on code execution at a specific address. Usually, debuggers insert a special instruction into the program to implement a code breakpoint. This lets you set an unlimited number of breakpoints. But this method does not work with code located in ROM or Flash. In these cases, the hardware-supported instruction breakpoints come in handy; however, there are only two of them. Unfortunately, instruction breakpoints appear to be useless because they generate a "prefetch abort" exception, which is not passed to the *__try/__except* handler or a debugger.

Register *TBREG* is for reading bytes from the trace buffer and *CHKPT0* and *CHKPT1* are for associating execution history in the trace buffer with instruction addresses. Several other debug registers are for communication with JTAG debugger and are not discussed here.

The Process ID (PID) register is not a debug register, but used when preparing addresses to be set in *DBRx* or *IBCRx*. Windows CE can run up to 32 processes, each occupying its own 32-MB address slot. The current process is also mapped to slot 0, which lets a DLL code section (located in ROM) access different data sections when DLL is loaded in several processes. ARM architecture provides PID as a direct and efficient support for such slot remapping. The value of the PID is equal to the address of the process slot. The CPU uses the high 7 bits (31:25) on the PID to replace the correspondent bits of virtual addresses when they are 0. The same operation has to be performed when preparing addresses for *DBRx* or *IBCRx* (see macro *MAP_PTR* in Breakpoint.h).

Reporting Data Breakpoints

I present here three straightforward ways to handle data abort exceptions generated when data breakpoints are triggered:

- Using an application debugger.
- Using a *__try/__except* construct.
- Writing a simple kernel debugger stub.

Application debuggers (such as EVC) handle data abort exceptions in any thread of the program under debug. They break execution and display the source line or instruction that triggered the breakpoint, display registers, local variables, and call stack.

However, the application debugger often cannot be used because a connection is not available or it's too slow. Also, the debugger cannot handle exceptions in a system process, such as device.exe (hosts drivers) or gwes.exe (hosts user interface).

"Alas, in Windows CE 4.x/PocketPC 2003, the kernel debugger must be loaded as a kernel module"

The second approach is to wrap code in *__try{}__except(Expression){}* exception-handling blocks. When exceptions happen within the *try{}* block, the system executes an *Expression* statement. I implemented the function *ExceptionHandler* in BreakPointSamples.cpp, which should be specified as the argument to *__except*. I call the *_exception_info* API to get useful information, such as exception code, address, and CPU registers. *ExceptionHandler* displays this information in message boxes (to simplify integration of this code into various applications). Unfortunately, a *__try/__except* construct can only handle exceptions coming from a thread, which executes code within the *try{}* block or functions called from within the *try{}* block. This is not a problem if you can insert *__try/__except* into source code for all suspect threads in your program.

When printing information about an exception, it's best to print the stack trace. Printing the stack trace on an ARM is more difficult than x86 because the EVC compiler can generate several different types of function prologs and does not have an option to produce consistent stack frames (see "ARM Calling Standard" in EVC help for details). Also, unlike the x86, which always pushes the return address to the stack when calling a function, ARM code moves return addresses to a register *LR*. Most functions usually start by storing the *LR* on the stack, but highly optimized code can keep it in any register. This means that on ARMs, it may not be possible to reliably reconstruct the stack trace without disassembling the code (which is beyond the scope of this article).

A Simple Kernel Debugger Stub

It is sometimes necessary to catch exceptions globally—in any thread of any process. The easiest way to achieve this is to register a DLL as a kernel debugger stub. I include here the minimal code (available electronically) capable of handling system-wide exceptions. In the days of Windows CE 3.0/Pocket PC 2002, you could register a regular user DLL as a kernel debugger stub and display exceptions in a regular message box (the whole code was just about 200 lines). Alas, in Windows CE 4.x/PocketPC 2003, the kernel debugger must be loaded as a kernel module. The problem is that a DLL such as this cannot link to any other DLL, even *coredll* (which provides most CE API and C/C++ runtime library functions). Consequently, I had to implement my own *sprintf*-like formatting routine as well as integer division-by-10 (both are normally imported from *coredll*). I also recycled my old HTrace library to write trace to a shared memory buffer, which can be displayed from a separate application. You can find the code in the SimpleKDStub directory. To run it, copy SimpleKDStub.dll and KDViewer.exe to the PDA and start the program. It loads the stub, which starts listening for exceptions. Once an exception is caught, it is printed to a shared buffer and displayed in the application. This tool is useful for data breakpoints and for catching other exceptions in any application on the PDA.

XScale Trace Buffer

The XScale architecture implements a powerful debugging feature—the trace buffer. When enabled, it collects a history of executed instructions. The trace buffer is just 256 bytes long (built inside the CPU itself), but stores the history as a compact sequence of 1- or 5-byte entries representing control flow changes (exceptions and branches). Each entry has a 1-byte message, which indicates the type of entry (exception, direct, or indirect branch) and the count of instructions executed since the previous control flow change. If this count exceeds 15, then a special roll-over message is stored. Entries for indirect branches include

```
; void TestTraceBuffer():
 ...
280121B0   mov       r1, #1
280121B4   mov       r0, #1
280121B8   bl        |SetDebugControlAndStatus| ; enable trace buffer
280121BC   add       r0, sp, #0x11, 28
280121C0   bl        |Test|  ; call Test
 ...
; void Test(int * p)
280121F0   mov       r12, sp
280121F4   stmdb     sp!, {r0}
280121F8   stmdb     sp!, {r12, lr}
280121FC   ldr       r0, [sp, #8]
28012200   bl        |Test1|  ; call Test1
28012204   ldmia     sp, {sp, pc}

; void Test1(int * p)
28012208   mov       r12, sp
2801220C   stmdb     sp!, {r0}
28012210   stmdb     sp!, {r12, lr}
28012214   ldr       r0, [sp, #8]
28012218   mov       r1, #0x7B
2801221C   str       r1, [r0] ; *p = 123 - triggers a data breakpoint
28012220   ldmia     sp, {sp, pc}
```

Figure 1: *Annotated disassembly listing for functions* TestTraceBuffer, Test, *and* Test1 *used to demonstrate the XScale trace buffer.*

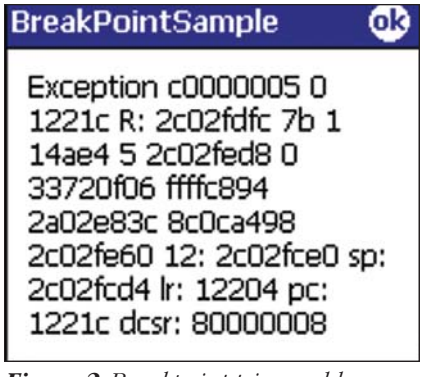


Figure 2: *Breakpoint triggered by function* Test1.

an additional 4-byte target address. The buffer may be configured to work in wraparound or fill-once mode. Wraparound is appropriate when waiting for an exception (as I do in this article). Fill-once mode (which generates a "trace-buffer full break" exception once the buffer is full) may be used to record all code execution continuously (however, I have not tried it yet).

The content of the trace buffer is extracted by reading the *TBREG* register 256 times (this also erases the buffer). The *CHKPTx* registers are used to get an address of a starting point for the reconstruction. Unfortunately, the buffer does not contain enough information to reconstruct the execution history without disassembling the executed code, counting instructions, and examining branches. Such a program is beyond the scope of this article. However, I included electronically the function *ShowTraceBuffer*, which simply displays the list of entries in the buffer in a series of message boxes. You can use this information, along with the disassembly window of the EVC debugger, to recover execution history prior to an exception. This may be a more powerful tool than a stack trace. Be aware that the trace buffer collects global execution information from all processes, the OS kernel, and interrupt handlers.

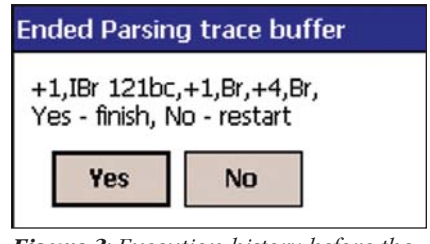


Figure 3: *Execution history before the breakpoint in function* Test1.

The function *TestTraceBuffer* in BreakPointSamples.cpp demonstrates using the trace buffer to record execution history. *TestTraceBuffer* sets a data breakpoint and enables the trace buffer, then it calls function *Test,* which calls *Test1,* which triggers the breakpoint. Figure 1 is an annotated disassembly listing for these functions. The exception raised by the breakpoint is displayed in a message box in Figure 2, where you can see the address of the instruction that triggered the breakpoint (value of register *PC=1221C*). Register *R0 = 2C02FDFC* is the address of the data and register *R1= B(123)*–the new value. Figure 3 displays the parsed trace buffer: *+1, IBr121BC, +1,BR,+ 4,BR.* This lets you reconstruct the execution history: The function *SetDebugControlAndStatus* executed one instruction after enabling the trace buffer, then returned to address *121BC* (in *TestTraceBuffer*), then one instruction was executed, then branch (to *Test*), then four instructions and branch (to *Test1*).

Further Improvements

A simple way to enhance the code I present here would be to print the module name and offset instead of the raw return address when printing exception information. A more difficult exercise would be to print the stack trace or enhance the trace buffer printing with a disassembler to fully reconstruct the execution history. A completely new direction would be to implement a continuous execution recording tool using a fill-once trace buffer. I may post bug fixes and improvements on my web site (http://forwardlab.com/).

Conclusion

XScale-based CPUs provide a powerful support for hardware-assisted debugging. Fortunately, it is not necessary to wait for application debuggers to provide access to all CPU features from the GUI. On Pentium-based systems, Visual Studio never managed to implement breakpoints on data reading or hardware breakpoints on local variables. Therefore, it is important for you to know the capabilities of the CPU and how to exploit them from an application. The tricks I present here may not be for everyday use, but every now and then, they can save hours (or days) of difficult debugging.

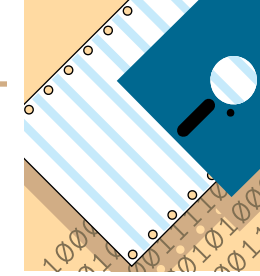
DDJ

Listing One

```
; SetDebugControlAndStatus writes (optionally) to
; Debug Control and Status Register (DCSR)
; and returns the original value of DCSR.
; parameters:
;   r0: flags to be set or reset in DCSR.
;   r1: mask - flags to be modified in DCSR, the rest is preserved.
; return value:
;   value of DCSR before the modification

    EXPORT    |SetDebugControlAndStatus|
|SetDebugControlAndStatus| PROC
    stmdb   sp!, {r2,lr}    ; save registers
    mrc     p14, 0, r2, c10, c0, 0 ; read DCSR to r2
    and     r0, r0, r1      ; r0 = r0 & r1 - clear flags not in mask
    bic     r1, r2, r1      ; r1 = r2 & ~r1 - leave flags not in mask
    orr     r0, r0, r1      ; r0 = r0 | r1 - combine flags
    cmp     r0, r2          ; compare new with original
    mcrne   p14, 0, r0, c10, c0, 0 ; write DCSR if flags have changed
    mov     r0, r2          ; prepare to return the original flags
    ldmia   sp!, {r2,pc}    ; restore the registers and return
```

DDJ



Pining for the Fjords

Michael Swaine

For some reason, all the topics in this month's column have something to do with death or legacy. Dark. And I haven't even seen Frank Miller's *Sin City* yet. I talk about an end-of-life language, two ex-Microsofties, a television Terminator, legacy code and coding legacy, and what comes after physics. To lighten things up, I allow some jokes to slip in. Fjordian slips.

It crossed my mind when these irreverent references were slipping into the text that some readers might consider them to be in poor taste. Monty Python jokes, such readers might say, are hardly appropriate in referring to matters of life and death, and none of the above belongs in a column about programming paradigms. And what are programming paradigms, anyway?

My three responses to such readers, if there are any among the *Dr. Dobb's* readership, are: (1) Good point. I hope you've mentioned your outrage over exploitation of death to Tom DeLay and Jesse Jackson. (2) I'm older than you and closer to death. I have a special dispensation to laugh about the subject. (3) May I suggest that you Google "tasteful humor" and after slogging through the depressing results then tell me that the phrase is not an oxymoron? (4) You're absolutely right. I'm so ashamed.

And now for something completely basic.

VB-not-net: Pushing Up Daisies

It's hard to believe that, nearly 50 years after John Kemeny and Thomas Kurtz democratized programming by creating the truly revolutionary programming language Basic, 30 years after Basic became the first consumer HLL available for a microcomputer (by a company then called "Micro-Soft"), and 29 years after two guys named Dennis Allison and Bob Albrecht decided that a magazine (then called *Dr. Dobb's Journal of Tiny Basic Calisthenics and Orthodontia: Running Light without Overbyte*) needed to be created to put a free and open version of Basic into the hands of microcomputer users — after all this time, we're still talking about Basic.

Anyway I am.

And I'm not alone. Microsoft is in the process of inflicting a new programming language implementation on its developers and, no doubt in response to a subliminal Gatesian mandate insinuated into the brain of anyone who drinks from the well on the Microsoft Campus, they are calling it Basic. Microsoft always has to have a Basic, like the Winchester widow had to keep adding onto her house, like sharks have to keep swimming, like Frank Miller needs to share his noir. On the day that Microsoft no longer has a product called Basic, Mount St. Helens will rain down ash on the Microsoft Campus to the tops of the cubicle walls and the tectonic plate beneath castle Gates will shift, sending Bill's house floating out Puget Sound to sea on the crest of a tsunami.

But Microsoft is as loose in what it calls Basic as I am in deciding what fits under the heading of programming paradigms. Visual Basic, or VB, is evolving into VB-dot-net, and VB-dot-net is arguably about as close to VB-not-net as VB-not-net is to Kemeny and Kurtz Basic, which is to say not very. And therein lies the rub that galls the kibe of the down-at-the-heels developer. Because Microsoft has informed the approximately six jillion VB-not-net developers that it will no longer support their EOL'd version of VB. Free support ended March 31 of this year; paid support will linger for another three years, or five if the U.S. Congress passes a special law.

But a lot of those developers claim that VB-dot-net is a substantially different development platform, and not one that they necessarily want to migrate all their code to. Thousands of VB-not-net developers petitioned Microsoft not to cut off their life support.

Microsoft's response: The not-net version of VB has passed on. It is no more. It has ceased to be. It's expired and gone to meet its maker. It's a stiff. Bereft of life, it rests in peace. Its metabolic processes are now history. It's kicked the bucket, it's shuffled off this mortal coil, rung down the curtain, and joined the bleedin' choir invisible. It is an ex-language implementation.

Or words to that effect.

Bring Out Your Dead Code

Enter the folks at REALbasic, who give VB's head three perfunctory taps with a golden hammer, wrest away the Fisherman's Ring, and immediately parade their language's palpability before the stunned conclave of VB-not-net developers.

It's too late to take advantage of the deal now, aren't I a big help, but REALbasic offered its cross-platform V-like Basic to VB-not-net developers for a price that is hard to beat — nothing. Hard to top that for chutzpah — using one of Microsoft's own take-no-prisoners competitive tactics against it. The deal was set to end on April 15, though. As of April 1, more than 10,000 VB-not-net developers had taken advantage of it.

REALbasic is not a clone of Visual Basic exactly, and although the RB folks have

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

made it as easy as they can to port VB-not-net programs to RB, it's still a job. One might find it a less onerous job than porting to VB-dot-net, though, because RB is much more philosophically in synch with VB-not-net than the dot-net version is. Which is certainly a strength but also arguably a great weakness. Because the philosophy in question is preobject-oriented, or at least subobject-oriented, a throwback to earlier programming models, like, well, like Basic. It could be argued, and Microsoft argues thus, that if these throwback developers had any gumption at all they'd want to move into the 21st century. Even though I am one of those throwback RB coders, I do get the point.

Maybe it is time to move on — although this old language does have such lovely plumage.

Life After Microsoft

A brief aside on an early Basic and a programmer's legacy: Among the many Basics that have been developed and/or sold by Microsoft, the Visual and the invisible (Excel Basic), the Quick and the dead (MS-BASIC), one of the best known is GW-BASIC. What is not so well known is what or who GW was. There is a fairly widespread belief that GW meant "Gee Whiz." This is not correct. Not does GW stand for "Gates, William." Unless authorities in positions to know are deluded, GW-BASIC honors early (single-digit employee number) Microsoft employee Greg Whitten, who presumably had something to do with its development.

If the name honors Whitten that's a lot more than fellow former-Microsoftie Joel Spolsky does in his essay "Two Stories" (http://www.joelonsoftware.com/articles/TwoStories.html/. Joel more or less takes credit for the dismantling of Microsoft's Application Architecture group, which he perceived as a bunch of out-of-touch Ph.D.s arguing about how many macros can dance on the head of a pin. The Application Architecture group was headed by Ph.D. Greg Whitten.

I don't know how fair Joel is being, but this is all ancient history. According to Greg Whitten's bio, he "developed Microsoft's common cross-language compiler and runtime technology, the company-wide object-oriented software strategy and the software architecture strategy for the Office, Back Office, and Windows product lines," which sounds pretty impressive to me. But maybe Greg should be judged on the basis of what he's doing now, which is serving as the CEO and chairman of the board of NumeriX. (He also collects and races vintage cars, but it's NumeriX that will allow me to segue into the bit about computer mathematics and Mathematica.)

NumeriX provides software to technical people in financial institutions to help them make buying and selling decisions regarding exotic derivatives. Risk assessment stuff, Monte Carlo methods. The company's client list is impressive, including the World Bank. I'm not qualified to judge the quality of their work: I don't know what an exotic derivative is, couldn't even tell you what a vanilla derivative is (although I do know what a Monte Carlo method is — and no, it doesn't involve fear, surprise, ruthless efficiency, and an almost fanatical devotion to the Pope). I do understand, though, that the company is packed with financial and mathematical PhDs, which is probably a more comfortable environment for Whitten than Microsoft back in the day. Perhaps NumeriX, rather than GW Basic, will define Greg Whitten's legacy.

What Comes After Physics

Every so often I check the web site of the Public Library of Science, a peer-reviewed, open-access journal published by a nonprofit organization committed to making scientific and medical literature a public resource. It's such a fine idea that I want to support it, even though so far most of the articles are on medical or biological topics that are obscure to me. But recently I found an article titled "Mathematics Is Biology's Next Microscope, Only Better; Biology Is Mathematics' Next Physics, Only Better," by Joel E. Cohen of the Laboratory of Populations, Rockefeller and Columbia Universities, based on his keynote address at the NSF-NIH Joint Symposium on Accelerating Mathematical-Biological Linkages (http://biology.plosjournals.org/).

Cohen isn't merely saying that mathematics will be important in making new discoveries in biology. He's saying that the challenges in biology will drive the development of new mathematics. Most intriguing to me is the need in biology to deal with multilevel systems, in which events happening at higher or lower levels can have consequences on the current level. When you deal with cells within organs within people in human communities in physical, chemical, and biotic ecologies, and causality can stretch across all levels in nonlinear ways — well, maybe you should read the article. I know I found it inspiring.

DDJ

Dr. Ecco Solution

Solution to "Optimal Farming," *DDJ*, May 2005.

1. Observe that the minimum circumscribing circle for a square having side $2L$ has radius $L\sqrt{2}$ and area $\pi\times2\times L^2$, so the extra area is $(\pi-2)\times2\times L^2$. This is divided equally among the four sides. So each side gets $(\pi-2)\times(L^2)/2$ extra area. In the design of Figure 1, the small squares have side lengths $2L$ where $L=1/4$. The circles of radius L have 10 extra sides having total area of $5\times(\pi-2)\times(1/16)$. The large square has $L=1/2$, so has $(\pi-2)/2$ extra area. The total cost is slightly less than $(13/16)\times(\pi-2)$.
2. For the case where the circles all must have the same radius, try the design where there are four squares each having side length $2L$. There are two illustrated in Figure 2. Then there are circles covering each of these squares. So the radius of the circumscribing square is $L\sqrt{2}$. (Note that the middle of the rectangle is $1-2L$ from each side square.) In addition, there is a circle having a radius $L\sqrt{2}$ whose center is the middle of the rectangle. That reaches the near-center corner of the corner square because that distance is (by the Pythagorean theorem) $\sqrt{((1-2L)^2+(1/2)^2)}=\sqrt{(1-4L+4L^2+1/4)}=\sqrt{(5/4+4L^2-4L)}$. The length is $L\times\sqrt{2}$. Squaring that gives $2L^2$. So $2L^2=4L^2-4L+5/4$, yielding the quadratic expression $2L^2-4L+5/4=0$. So, $L=4/4+/-\sqrt{(16-10)}/4=1-(\sqrt{6})/4$.

M.S. Gopal contributed greatly to these solutions.

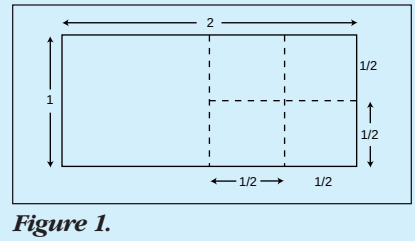


Figure 1.

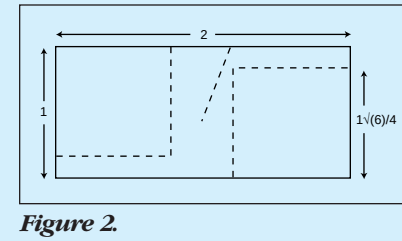
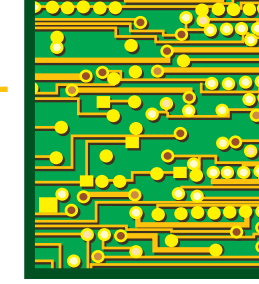


Figure 2.



Security Remeasured

Ed Nisley

Cryptography revolves around the keeping of secrets: knowing things that others should not. Although the only certain way of keeping a secret is to never tell it to anyone else, useful secrets require at least two people in the know. Keeping everyone else *out* of the know is the hard part.

Understanding the hardware and algorithms of a particular crypto scheme is easy compared to the challenge of actually deploying a secure system. Classic security can lock down point-to-point communications between a few relatively secure locations. We're discovering the inadequacy of those techniques applied to widely deployed embedded systems.

In recent months, I've read of several incidents that show how secrets sneak out of their crypto containers. Embedded systems with long lifetimes must keep secrets from their users, so these attacks reveal only the leading edge of the wedge.

Cracking RFID

In my January column, I related the story of how to get a month's free gas from a stolen SpeedPass token. Shortly after that column went read-only, a group of researchers from Johns Hopkins and RSA Laboratories described their SpeedPass crypto crack: You can now get a month's free gas without even touching the victim's keyring.

The SpeedPass payment system lets you buy gasoline by simply holding an RFID token near a reader in the fuel pump. Each token contains a Texas Instruments Digital Signature Transponder (DST) chip powered by the radio-frequency energy from the reader, so it can operate without batteries inside a sealed housing. Each DST has a unique, hard-coded 24-bit serial number and a programmable 40-bit signature (essentially a crypto key), a radio transponder, some special-purpose crypto hardware, and barely enough compute power to make it all work.

The 24-bit serial number identifying the token links into an account entry in ExxonMobil's customer database, which contains the 40-bit signature for that token. The RFID reader and token engage in a challenge-response handshake to verify that both sides know the same signature without exposing it to public view. If they agree, the reader activates the pump, the charge appears in your database entry, and away you go.

In pragmatic terms, this might not be a whole lot faster than swiping a credit card through the adjacent card reader, but it's attractive enough to be marketable. You can also buy junk food and other necessities using a SpeedPass, which might be faster than cycling your wallet at the head of a line at the counter.

The 40-bit signature is the system's only secret, despite a few other flourishes. Because formulating the challenge-response handshake requires knowledge of the signature, there should be no way to extract the signature without knowing it in the first place. Note that the customer does not know the secret, and in fact, the SpeedPass system design depends on that lack of knowledge.

However, because the customer possesses the hardware containing the signature, the initial attack can take as long as required. It turns out that extraction uses off-the-shelf hardware, a dash of cryptanalysis, and computational brute force. I won't repeat the researchers' analysis here, other than to observe that it's a great example of what notable experts can accomplish with a gaggle of talented graduate students.

The end result is a hardware system that can extract the 40-bit digital signature from any SpeedPass token *without* first knowing the signature, given just two challenge-response transactions. A token can handle eight transactions per second, so acquiring the data is essentially instantaneous and, because it's RF, doesn't require physical possession of the token. A few hours of computation on a special-purpose parallel processor, which costs a few kilobucks, suffice to crack the crypto and produce the signature.

The researchers project that some engineering and a touch of Moore's Law can increase the range to a few feet, stuff the machinery into an iPod-size case, and shrink the cracking time to a few minutes. The attack scenarios include sniffing SpeedPass tokens at a valet parking station (just

*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.*

wave the victim's keyring near your pocket) and walking the length of a subway car or mall holding a neatly wrapped package containing a big antenna.

The SpeedPass system's fraud-detection logic trips on excessive purchases, impossible locations, or atypical usage. However, if you harvest a few thousand tokens and use each one exactly once, you can probably get free gas for a long, long time.

While longer keys and tougher crypto may delay the inevitable cracks, the basic principle seems to be true: You cannot keep a secret from someone if you give them the secret. Seems obvious, doesn't it?

As is typical of widespread embedded systems, quickly replacing or updating the entire SpeedPass infrastructure to use better token hardware is essentially impossible. The only short-term defense against this type of attack involves wrapping the token in aluminum foil to shield it from cracked readers.

Designers of always-on devices, take note!

Cracking Trust

My February column discussed the mechanics behind the Trusted Platform Module (TPM) found in some recent laptops and desktops. Several readers pointed out that the "Trusted" part of the name has a peculiarly Orwellian definition: In actual fact, many software and media companies do not trust their customers. The companies depend on hardware to increase the effort required by customers who might otherwise easily steal their software, data, or (shudder) music.

The essential TPM feature is a secure hardware-storage mechanism, typically a single-chip micro or a few chips within an armored package, holding crypto keys, digital signatures, or hash values. Well-validated protocols allow external access only by trusted programs with the appropriate secrets of their own. You can't even examine the information without destroying the TPM, quite unlike secrets stored on disk.

Software running on the PC can validate itself using hashes stored within the TPM, authenticate itself to programs running on external servers using digital signatures from the TPM, then download and store data that requires further crypto keys for access. As long as the secrets stored within the TPM remain unknown to the PC's user, the whole chain of trust from musical source to hard drive remains unbroken.

Sounds iffy to you, too, huh?

To build a system using Trusted Platform Modules, manufacturers must have access to documentation and sample parts long before production begins. *The Inquirer* reports that Infineon will not support small manufacturers or system integrators, claiming that they will supply TPMs only to "qualified" customers. The story doesn't go into much detail, so we're left with suppositions rather than facts.

The researchers who cracked the SpeedPass had some support from Texas Instruments in the form of development kits and sample DST tokens, as well as their own SpeedPass tokens and car keys.

"The 40-bit signature is the system's only secret"

They did not have access to the DST's internal logic diagrams or other proprietary information, and in fact, discovering how the DST worked formed a major part of the effort.

Infineon may believe in "security through obscurity" or there may simply be licensing issues that we don't know about. In any event, if the security of the whole Trusted Platform Module infrastructure depends on keeping the documentation out of the hands of the bad guys, it doesn't stand a chance.

Cracking The Wall

Perhaps the single most obvious (and most touted) feature of Linux systems is their immunity to Windows security flaws. Linux and GNU software may present a compelling TCO justification, provide generally higher reliability software, and reduce the time to get bugs fixed, but security seems to be driving a broad-based change of opinion in their favor.

One unfortunate side effect reduces Linux system security to sound bites: "Linux is immune to viruses" and "Crackers don't bother Linux systems" and so forth. While Linux eliminates many of the common exposures, it cannot completely solve the problem.

One member of the Mid-Hudson Linux User Group noticed that his system had begun behaving strangely and asked for advice. His first post to the LUG's mailing list was titled "Have I been cracked?" and noted that:

> I don't recall making an account called 'systens', but apparently, someone ssh'd into it from 200.96.xxx.yyy. 'host' returns this info about that address…
>
> yyy.xxx.96.200.in-addr.arpa domain name pointer 200-096-xxx-yyy.cpece7021.dsl .brasiltelecom.net.br.
>
> brasil telecom? uh oh.

Mainstream Linux distros install and enable software firewalls by default during installation. In this case, however:

> I am running a firewall through my physical router. The inbound ports I open are for ssh, http, https, smtp, pop, 8080, and 81 for apache tomcat, ftp, and dns. […] I'm not running a software firewall on the box itself though.

By default, hardware firewalls block incoming packets that are not related to previous outbound messages. If you are running a server on your system, however, the firewall must pass incoming connection to a particular port directly to the server, which means the server is directly connected to the Internet. Any security flaw in the server provides a direct link into the system:

> Now that you mention it. I had a few CMS packages running on there. Namely, tikiwiki, drupal, and blog::cms. I locked down one of the tikiwiki instances […]. The other instance was open to the public for use and anyone could use it - not as admin, but with rights to modify the wiki and add forum entries, etc.

Tiki systems lets users create and update web pages from their browsers, which means anyone with a web browser can change files on the system. Any security flaw provides an opening:

> Saw this vulnerability in the tikiwiki web site of mid January. […] The vulnerability, initially, lets a user get a sort of shell on the server under the web server user. From then on, it is just a matter of time.

With the Web and tiki servers exposed to the Internet, your "users" can, indeed, be anyone:

> I bet that was it. I'll check my logs tonight when I get home. The 'systens' user account apparently was created on the 23rd - just one short week after this flaw was apparently reported.

Once an intruder has gained access to your system, becoming root isn't all that difficult and after the intruder is root, all manner of things become possible. If you don't use secure passwords on all your internal accounts, things become even more interesting:

> I ran across this in my "/var/log/auth.log" file…
>
> Feb 19 22:13:03 debian sshd[3020]: Accepted password for root from 192.168.1.1 port 3064 ssh2
>
> This is curious because 192.168.1.1 is my router. […] Is this just a bug in the router (I DO have NAT enabled) or something more I should perhaps worry about?

About a year ago, I described the process of cracking a router and uploading new firmware. I also observed that most users never change the admin account's password. That turns out to be a necessary, but not sufficient, security step:

> My firmware is up-to-date, and I don't have remote access turned on. […] However, I did use the same password for one of my accounts as I did for the router setup. So, in theory if that rootkit could crack passwords it could also allow access to the router.

The consensus advice for cleaning up after an intrusion boils down to two steps: reformat the hard drive and reinstall everything from scratch. You cannot assume anything about the compromised system — any command or program may do *anything*, including spoofing innocent return values.

In fact, you cannot assume anything about the status of any systems connected to the local network. In this case, the compromised router provides a direct link to the internal network, so restoring the compromised system wouldn't eliminate the vulnerability.

The lesson to be learned from this adventure is the inadequacy of simply keeping the patches to an Internet-facing system up to date. You must also monitor the system logs, become familiar with "normal" operation, and track down any anomalies to their source. That this level of involvement far exceeds the abilities or interests of most PC users, alas, goes without saying.

A Windows XP SP1 system without a firewall will be compromised in minutes, while a firewall completely eliminates incoming attacks. A firewall with open ports requires meticulous system security practices on the systems exposed to the Internet. In the end, however, firewalls and up-to-the-minute patches form just the first line of defense. Attention to detail must provide defense in depth.

Pop Quiz: What do we do with always-connected embedded systems with no user interface? *Essay*: Describe the user manual's section on network security.

Reentry Checklist

More on the SpeedPass RFID tag cracking adventure at http://rfidanalysis.org/. You should definitely read their preliminary research paper at http://rfidanalysis.org/DSTbreak.pdf, which does not provide quite all the details required to crack SpeedPasses on your own.

The Inquirer article on Infineon's TPM policy is at http://www.the-inquirer.com/?article=21113. Everything Infineon has to say about its TPMs, at least to us, seems to start at http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_ov.jsp?oid=29049.

An overview of Digital Rights Management and online music from a Canadian perspective at http://www.pch.gc.ca/progs/ac-ca/progs/pda-cpb/pubs/online_music/tdm_e.cfm, which uses a different definition of "TPM" than you see here.

Magnatune carries music released under the Creative Commons license at http://www.magnatune.com/, entirely without DRM. Streamtuner for Linux simplifies access to a myriad audio streams at http://www.nongnu.org/streamtuner/, which is completely different from whatever's offered on the stub page at http://www.streamtuner.com/.

Writeups of the Linksys router flaw are at http://secunia.com/advisories/11754/ and http://www.wi-fiplanet.com/news/article.php/1494941. An experiment measuring "system time to live" on the Internet is at http://www.avantgarde.com/xxxxttln.pdf. Kevin Mitnick consulted on the study, should that name ring a bell.

Thanks to Alan Snyder, Sean Dague, Renier Morales, and MHVLUG for allowing me to slice up their threads. The original archives are at http://www.mhvlug.org/.

DDJ

Drive-By Spyware & Other Horrors

Jerry Pournelle

The Spyware menace has gone beyond all reason. Spyware costs time and money. It costs you directly, and it destroys Aunt Minnie's confidence in the Internet. Something must be done; indeed something will be done because if things go on as they are, the Internet itself is doomed.

And yet, despite the increasing Spyware/Adware/Malware assaults, there are spyware companies out there with lawyers sending warning messages to anyone who calls their malware by its right name. See, for example, the story at http://www.ahbl.org/notices/iSearch.php.

We have tools that can partially protect us against this plague, but they don't entirely work, and they take time and effort to use. Meanwhile the lawyers are having a field day defending the rights of their clients to invade and take over your computer. They claim that you have agreed to let them do it, and they have every right.

Here is a typical "license agreement" that supposedly sane users have in theory "accepted":

> 2. Functionality—Software delivers advertising and various information and promotional messages to your computer screen while you view Internet web pages. iSearch is able to provide you with Software free of charge as a result of your agreement to download and use Software, and accept the advertising and promotional messages it delivers.
>
> By installing the Software, you understand and agree that the Software may, without any further prior notice to you, automatically perform the following: display advertisements of advertisers who pay a fee to iSearch and/or it's [sic] partners, in the form of pop-up ads, pop-under ads, interstitial ads and various other ad formats, display links to and advertisements of related web sites based on the information you view and the web sites you visit; store nonpersonally identifiable statistics of the web sites you have visited; redirect certain URLs including your browser default 404-error page to or through the Software; provide advertisements, links or information in response to search terms you use at third-party web sites; provide search functionality or capabilities; automatically update the Software and install added features or functionality or additional software, including search clients and toolbars, conveniently without your input or interaction; install desktop icons and installation files; install software from iSearch affiliates; and install Third Party Software.
>
> In addition, you further understand and agree, by installing the Software, that iSearch and/or the Software may, without any further prior notice to you, remove, disable or render inoperative other adware programs resident on your computer, which, in turn, may disable or render inoperative, other software resident on your computer, including software bundled with such adware, or have other adverse impacts on your computer.

I submit that no one in his right mind has ever agreed to this; that the only way it was "agreed" to was by stealth, not through anything like informed consent. There may be, out among the Aunt Minnie's of this world, one or two who actually saw something like this and "agreed"; but how many *DDJ* readers would accept such a thing?

Drive-By Spyware

My own case is illustrative. I had a report from Associate Editor Dan Spisak on his not very successful attempts to remove the iSearch Toolbar and a number of other infections from a friend's machine. He tried everything, and in the course of his efforts discovered that, while it goes without saying that Internet Explorer was hijacked, even the Firefox browser was affected.

I collected notes on this and other spyware subjects in OneNote on my Tablet-PC, then started another column section on Bill Gates's recent speech at the Governor's Conference. When I did a Google search for a particular quote to use, I found one likely source at a place called "Study World." Fair warning: If you want to go look there, set your browser security level to HIGH and don't agree to anything.

When I went to that location, up popped a warning from Microsoft that I didn't read closely, and in a moment of sheer madness I clicked OK. Microsoft Anti-Spyware instantly popped up to warn me that something was trying to infect my system. Other warnings came thick and fast. Meanwhile, though, my Internet Explorer browser changed home pages. Popup advertisements of every kind began to appear. My system was in real trouble.

Microsoft Anti-Spyware said it was blocking this and that (about six messages, all stacked). I told it in each case to block the stuff, and I closed the browser. Eventually that flurry of messages stopped, but when I ran Microsoft Anti-Spyware, it found I was infected with WinTools, Toolbar Websearch, Network Essentials Browser Modifier, and CYDOOR adware. Microsoft Anti-Spyware offered to remove them, trundled, and then said it had removed them. Then Microsoft Anti-Spyware wanted me to reset the machine.

I wasn't sure I wanted to do that just yet. Suspiciously, I ran Microsoft Anti-Spyware again. It produced precisely the same result, finding the same infections. I ran AdAware and Spybot Search and Destroy. They didn't find anything wrong at all. Clearly, the infection had managed to bypass or compromise all my antispyware tools.

Now I was sure I was in trouble. I quickly opened a command window and used XCOPY to copy off to a thumb drive all the files in the places I keep documents, using the /e/s/d/y switches to get only those I hadn't backed up recently. (I keep batch files for just that purpose.) With that done, I was ready to battle for possession of my machine.

Jerry is a science-fiction writer and senior contributing editor to BYTE.com. *You can contact him at jerryp@jerrypournelle.com.*

Removing WinTools

The first move was to use Microsoft Anti-Spyware one more time and this time reboot. As I suspected, that did nothing at all: Although Anti-Spyware was very unhappy, WinTools was still in there, complete with the directory Program Files/Common Files/Wintools that WinTools creates and Microsoft Anti-Spyware thought it had deleted. Deleting that directory does nothing until you get the actual program that regenerates it; this is considerably harder because it hides deep in your file system in another directory entirely (in my case it was hiding in the System 32/DRIVERS subdirectory but infections use different hiding places). In fact, eliminating the source generator for the infection files is beyond the capabilities of any automatic program I can find.

Time to go to hell — that is, the web site PCHell (http://www.pchell.com), particularly to http://www.pchell.com/support/wintools.shtml where there are complete instructions for getting rid of the Wintools infection. Well, practically almost: I would never have got rid of this thing without the PCHell folks, and they have my gratitude, but even their instructions didn't do it all. They also led me to the wonderful HijackThis program (http://www.spychecker.com/program/hijackthis.html) which, despite the ominous name, is one great program. While you are thinking about it, go download that program so that it is on your computer. You may never need it, but if you do need it you will want it badly.

The solution to exorcising Wintools involves rebooting in Safe Mode. Once in Safe Mode, open a command window (or Norton Commander) and eliminate any directory called Wintools. Then edit the registry to remove every reference to Wintools. That done, use HijackThis several times. HijackThis will find things you don't want removed — such as the Google toolbar — and offer to do other things you don't want done because its goal is to get your system as close to the default registry configuration as possible; meaning that you want to employ some intelligence when using the program. On the other hand, better safe than sorry. You can always reinstall things you want to keep if you have accidentally eliminated them.

When you run HijackThis, you get a list of registry entries the program doesn't care for. There is an option to check items on that list one at a time and ask for more information. One item might be "This is a change from the default home page for your browser." Another might be a reference to the Yahoo Toolbar. In each case, HijackThis has an option for "fixing" the problem. The fix in general will be to eliminate the registry key, or to restore it to the Windows XP default value.

I let HijackThis fix everything I didn't understand. Once I had used Regedit and HijackThis but before I left Safe Mode, I ran AdAware and Spybot Search and Destroy. Neither found anything but cookies, which Spybot wanted badly to remove for me. Then I ran Microsoft Anti-Spyware, and lo! it found references to CYDOOR, an advertising robot. (CYDOOR has a web site; you can go there and see what they claim to be doing. It won't say anything about stealth infections. Before I went over there I made sure my browser security setting was "HIGH" and I wished there was a level above that; I'd call it "PARANOID.")

I let Microsoft Anti-Spyware remove the CYDOOR references, then I went through the registry searching for Wintool — and found several more references. I think they were pretty well harmless by then, but I deleted them all, and then did a search for CYDOOR, but found nothing. Then I ran HijackThis one more time.

This time, HijackThis found nothing I didn't understand. Neither did Microsoft Anti-Spyware.

All was well. I shut down the system, turned it off, counted to 60, and brought it up again. All is still well according to freshly reinstalled copies of AdAware, Spybot Search and Destroy, and Microsoft Anti-Spyware. If any spyware lurks in here, it's not doing anything. HijackThis lists all running services, and there are none I don't expect.

Trusting the Enemy

Many Adware/Spyware/Malware programs will direct you to a "removal site," where you can download a program that, they promise, will remove their Adware and restore your system to its pristine condition.

To do that you will have to run an executable program provided by a company that snuck up on you and installed its ware by stealth.

If you think this is a good idea, please contact me. I have a bridge I want to sell you.

Lessons Learned

First and foremost: When your system tells you something, listen. In this case, I was actually warned that something wanted to install, and I let it, thinking that Norton Anti-Virus and Microsoft Anti-Spyware would prevent any real problems. They didn't. By the time Anti-Spyware got on the job, the damage had been done — and Norton didn't do a thing.

Second: Least done, soonest mended. If you think you've been infected, stop what you are doing and deal with it. Don't give it a chance to do any more damage. Pull your Ethernet connection plug and start disinfecting.

Third: Seriously consider using Firefox rather than Internet Explorer. I say this although there is evidence that some malware understands Firefox. I don't make this an absolute because this malware didn't exploit a hole in Internet Explorer at all. It talked me into letting it install. I am not sure what I thought I was allowing it to do, but I did okay something. I have no reason to believe Firefox can protect me against stupidity. If you let an executable program download and run, browser theft is likely to be the least of your problems.

"When your system tells you something, listen"

Fourth: If you are considering participating in various software and music swapping schemes that give any kind of control over your system to the scheme, don't do it. Malware and viruses and worms, o my! ride in with those file swapping schemes. There ain't no such thing as a free lunch. I got bit by a drive-by browser infector. Bringing in file swapping schemes practically invites infection.

Fifth: Nothing finds it all. The Microsoft Anti-Spyware program is pretty good, but you'll also want to keep AdAware and Spybot Search and Destroy. Don't run them at the same time.

Finally: Well, when you come down to it, it cost me an hour. In my case, it gave me something to write about. But it wouldn't have happened if I'd been doing all this on a Mac or a Linux box.

Windows XP 64-Bit Edition is Coming!

Earlier this month, Microsoft released Release Candidate 2 of Windows XP 64-bit Edition for AMD and Intel processors along with a confirmation that the product would be released to retail stores by the end of April. If you happen to have an AMD 64-bit processor, such as the Opteron or Athlon 64, then this is the release you will want to experiment with before the retail version appears. We installed it on our NForce3 250-based motherboard without incident. More drivers were included in this release of the OS as it managed to install drivers for the Marvell gigabit Ethernet chipset onboard this time along with chipset drivers. (64-bit XP Release Candidate 1 didn't recognize this chip.)

We can also successfully report that going to the Start Menu and selecting Windows Update now works as it should without giving cryptic error messages. ATI and NVIDIA both have recent builds of drivers available for their video cards and motherboard chipsets for XP 64. Give the current release candidate a try if you have been waiting for a working version of the OS to test. This time it works pretty well.

Winding Down

I had expected the Game of the Month to be the new release of Sid Meier's *Pirates!*. I very much enjoyed the game on my early Macintosh, and was disappointed when I couldn't get any of the PC or Windows versions to run well. Consequently, when the new one came out, I leaped at it.

Alas, it has been a disappointment to me, largely because there is no game speed control. Now I realize I can jigger one up. I can run it on a slow machine and employ one of those programs that waste cycles, but I don't really want to do that; perhaps it's mere funk. For me, though, the game plays too fast, so that it's more like a shooter than the delightful combination humor/role playing game that the original *Pirates!* was.

Clearly, my view isn't shared by all. The game has many excellent reviews, and indeed, except for the unchangeable too fast game speed, I found little to dislike. It retains much of the flavor of the old game, but with better graphics. There's a lot to like about it, but I find that it tires me to play it for long. Ah well, back to *Everquest II*, except that I find I am putting too much time into that.

The first computer book of the month is Kathy Jacobs and Bill Jelen's, *Life on OneNote* (Holy Macro Press, 2004). If you are a TabletPC user, or thinking of becoming one, you'll want Michael Linenberger's *Seize the work Day: Using the TabletPC to Take Total Control of Your Work and Meeting Day* (New Academy, 2004). It has a wealth of examples of TabletPC applications and how to use them. If you are building your own equipment, you already know you need Bob and Barbara Thompson's *Building the Perfect PC* (O'Reilly & Associates, 2004). You'll also want their *PC Hardware Buyer's Guide* (O'Reilly & Associates, 2005) to help you choose components.

DDJ

Greg on Software Books

Gregory V. Wilson

The older I get, the more I find myself trying to make sense of things. Crazy, I know, but I don't just want my code to work anymore—I want to understand how it works from top to bottom, and how it fits into the grand scheme of things.

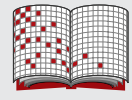
Judging from this month's books, I'm not the only one who feels this way. On the top of the list is Joel Spolsky's *Joel on Software*, which collects some of the witty, insightful articles he has written over the past four years. If you're a developer, Spolsky's weblog is a must-read: His observations on hiring programmers, measuring how well a dev team is doing its job, the API wars, and other topics are always entertaining and informative. Over the course of 45 short chapters, he ranges from the specific to the general and back again, tossing out pithy observations on the commoditization of the operating system, why you need to hire more testers, and why NIH (the "not-invented-here" syndrome) isn't necessarily a bad thing. Most of this material is still available online, but having it in one place, edited, with an index, is probably the best $25.00 you'll spend this year.

Budi Kurniawan and Paul Deck's *How Tomcat Works* is a much narrower book, but seems to be driven by the same need to make sense of things. The book delivers exactly what its title promises: a detailed, step-by-step explanation of how the world's most popular Java servlet container works. The authors start with a naïve web server that does nothing except serve static HTML pages until it's told to stop. From that humble beginning, they build up to a full-blown servlet container one feature at a time. Each time they add code, they explain what it's doing and (more importantly) *why* it's needed. Their English is occasionally strained, and there were paragraphs I had to read several times to understand, but this book is nevertheless an invaluable resource for servlet programmers who want to know more about their world.

John Goerzen's *Foundations of Python Network Programming* is superficially different, but at a deeper level, very similar. Where Kurniawan and Deck look at one way to handle one protocol, Goerzen looks at how to handle several common protocols, including HTTP, SMTP, and FTP. Goerzen also doesn't delve as deeply into how servers work, concentrating instead on how to build clients that use these protocols.

The similarity lies in the approach. As with *How Tomcat Works*, Goerzen builds solutions to complex problems one step at a time, explaining each addition or modification along the way. He occasionally assumes more background knowledge than most readers of this book are likely to have, but only occasionally, and makes up for it by providing both clear code and clear explanations of why this particular function has to do things in a particular order, or why that one really ought to be multithreaded. I've already folded down the corners of quite a few pages and expect I'll refer to this book often in the coming months.

DDJ



Joel on Software
Joel Spolsky
APress, 2004
362 pp., $24.99
ISBN 1590593898

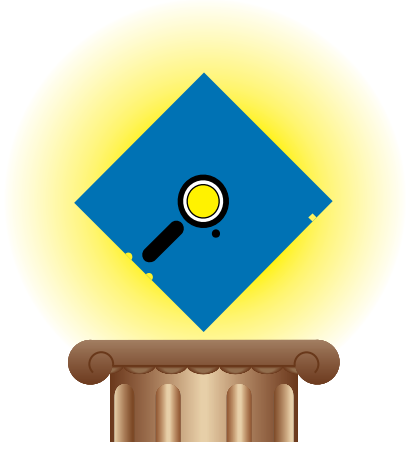
How Tomcat Works
Budi Kurniawan and Paul Deck
BrainySoftware.com, 2004
450 pp., $49.99
ISBN 097521280X

Foundations of Python Network Programming
John Goerzen
APress, 2004
512 pp., $44.99
ISBN 1590593715

Programmer's Bookshelf

Submissions to Programmer's Bookshelf can be sent via e-mail to editors@ddj.com or mailed to *DDJ*, 2800 Campus Drive, San Mateo, CA 94403.

Greg is a DDJ *contributing editor and can be contacted at gvwilson@ddj.com.*



Electric Cloud has sponsored the creation of the GNU Make Standard Library (GMSL), aimed at providing a common set of tools for all users of GNU Make. The GMSL includes list and string manipulation, integer arithmetic, associative arrays, stacks, and debugging facilities. The GMSL is released under the General Public License and is hosted on SourceForge. Electric Cloud also offers Electric Cloud 2.1, designed to reduce build times by distributing the software build in parallel across scalable clusters of servers.

Electric Cloud Inc.
2307 Leghorn Street
Mountain View, CA 94043
650-968-2950
http://gmsl.sourceforge.net/

Pantero Software is designed to create rules and services that deliver valid data in integration projects. You use graphical tools to import data schemas or models, map data from one representation to another, define rules to ensure validity and consistency, and define error-handling behavior—with all of it captured as metadata. The Pantero runtime software implements these rules as web services or Java controls. Pantero Release 1.3 supports Eclipse and the JBoss application servers as well as software from BEA and IBM.

Pantero Software
300 Fifth Avenue, Suite 630
Waltham, MA 02451
781-890-2890
http://www.pantero.com/

The 7.2 release of Amzi! Prolog + Logic Server incorporates the Eclipse 3.0 IDE to provide editing, debugging, and project management for logic programs, embedded logicbases, and remote logicbases on web servers. Amzi! 7.2 adds graphical icons for breakpoints indicating the four debug ports: call, fail, redo, and exit. In conjunction with the call stack, this function illustrates advanced logic programming features such as backtracking, recursion, and unification. Amzi! is available on Windows, Linux, Solaris, and HP/UX.

Amzi! Inc.
47 Redwood Road
Asheville, NC 28804
828-350-0350
http://www.amzi.com/

The EngInSite PHP Editor is an IDE that automatically recognizes public and private functions of the class. EngInSite PHP Editor is "HTML-aware" and comes with CSV support. Its architecture emulates HTTP server behavior in the Editor's output window. It also offers a breakpoint option and an option to upload PHP scripts to a server. SFTP, SSH1, and WebDAV are supported as well as FTP: The program connects to a remote server on its own, automatically uploading only new or changed files.

LuckaSoft
Anholter Strasse 2B
D46395 Bocholt, Germany
+49 2871 233 8 01
http://www.enginsite.com/

Recursion Software's C++ Toolkits, a collection of C++ class libraries, has added support for Sun Microsystem's Solaris 10 operating system, along with Linux, AIX, Tru64, HP-UX, IRIX, Red Hat Linux, Mac OS, VxWorks, Windows, PocketPC, and Suse/Linux. The C++ Toolkits enable development of multithreaded distributed applications, and include advanced class libraries for developing and deploying transactions in distributed and service-oriented architecture (SOA) environments.

Recursion Software Inc.
2591 North Dallas Parkway, Suite 200
Frisco, TX 75034
800-727-8674

ActiveSMS is an ActiveX DLL that lets you send and receive SMS through GSM/GPRS terminals. It handles voice calls and the phonebooks of both SIM and terminal. Integral into any application that supports COM technology, ActiveSMS provides an interface that exports "events" and "methods" to handle reception and forwarding of text, Flash, or binary messages, as well as multiple forwarding, voice call handling, and status report handling. It supports serial connections, IrCOMM, IrDA, USB, and Bluetooth and is able to communicate with the terminals both in "PDU mode" and in "Text Mode."

Net Sphaera S.n.c.
Via Torre Della Catena, 150
Benevento, Italy
http://www.activesms.biz/eng

Helixoft has released VBdocman .NET 2.0, a Visual Basic .NET Add-In for generating technical documentation from VB.NET source files. It parses source code and automatically creates table of contents, index, topics, cross references, and context-sensitive help. Users can create their own formats of output documentation: The predefined output formats are Help 2 (Microsoft help technology used in Visual Studio .NET), CHM, RTF, HTML, and XML.

Helixoft
Tomasikova 14
080 01 Presov, Slovakia
http://www.vbdocman.com/net/

IT GlobalSecure has updated SecurePlay Version 2.1, its multiplayer state engine. SecurePlay implements a suite of cryptographic protocols to stop many kinds of cheating and piracy, and offers a programming interface for multiplayer, networked game development. SecurePlay 2.1 includes integrated networking and interoperability between Java and C++ in Windows and Linux. You receive a complete copy of the documented source code in Flash, Java, J2ME, or C++, with a PS2 release forthcoming.

IT GlobalSecure Inc.
1837 16th Street NW
Washington, DC 20009-3317
202-332-5878
http://www.secureplay.com/

Graphics & Scripting Tools has announced Vector Graphics ActiveX, a fully fledged vector graphics platform for incorporating 2D vector graphics into an application-development cycle. The component is compatible with Visual C++, Visual Basic, and Delphi development tools and is designed to create professional-quality technical drawings, illustrations, charts, and diagrams. By establishing links between graphic shapes and real objects, the developer can connect to OPC servers to watch and modify processes in real time.

Graphics & Scripting Tools LLC
Kosm Strasse 10
394055 Voronezh, Russia
http://www.script-debugger.com/

DDJ

The Hanging Algorithm

"The prospect of hanging concentrates the mind wonderfully."
—Samuel Johnson

In his columns in *Scientific American* and in a book titled *The Unexpected Hanging and Other Mathematical Diversions*, Martin Gardner explores an intriguing paradox. It first saw print in 1948 and has been cast in many forms, involving surprise inspections, class-A blackouts, pop quizzes, hidden eggs, and card tricks. Patrick Hughes and George Brecht devote 20 pages of their book, *Vicious Circles and Infinities* to the paradox.

Although the paradox has been richly analyzed, there is one approach that I've never seen. Perhaps I missed it, or perhaps the approach I have in mind is just wrong. But for what it's worth, I thought I'd present my algorithmic analysis of the "Paradox of the Unexpected Hanging."

The paradox: On a certain Saturday, a man is sentenced to be hanged. The judge, known to always keep his word, tells him that the hanging will take place at noon on one of the next seven days. The judge adds that the prisoner will not know which is the fateful day until he is so informed on the morning of the day he is to be hanged.

The prisoner is elated, convinced that he will not be hanged. He reasons as follows: If he is alive on Friday afternoon, he will know that he is to be hanged on Saturday. But this contradicts the judge's assertion that he will not know his hanging day in advance. So his execution cannot be scheduled for Saturday. Therefore, Friday is the last day on which he can be hanged in keeping with what the (truthful) judge has said. But this means that if he is alive on Thursday afternoon, he knows at that point that he will be hanged on Friday— because Saturday has been conclusively eliminated. And by recursion, the prisoner reasons that he cannot be hanged on any day of the week. He is serene.

Thursday comes around and he is informed that he is to be hanged that day— a completely unexpected hanging. The judge spoke the truth. Where did the prisoner's reasoning go wrong?

It is entertaining to see the subtle ways in which some have gone wrong in trying to unravel the paradox. It is not, as some think, trivial. The judge's statements do not appear to be contradictory, the prisoner's reasoning seems to be perfectly logical, and yet something must be wrong with one or the other. We are tempted to suspect the judge of predicting something impossible, but then it turns out to be true. What's the resolution?

I think that the difficulty arises from confusing a prediction with a principle. The judge is not merely predicting that the prisoner will be surprised, he is saying that it is impossible for the prisoner to know the date of his execution in advance. This is a stronger claim and cannot be demonstrated by a single example.

In effect, the judge is claiming that there exists an algorithm for execution-day selection, and asserting that the prisoner cannot in principle determine the outcome of the algorithm early.

So can there be such an algorithm? I say no. Clearly the algorithm can't be something like "Choose Thursday." If that were the algorithm, the prisoner would know that Thursday was the day. So the algorithm must be probabilistic, such as "Choose Monday with probability 1/4, Tuesday with probability 1/4, or Wednesday with probability 1/2."

If you're thinking that there could be several algorithms like "Choose Thursday" and "Choose Wednesday," and that the judge could choose one, you're just multiplying algorithms, because then you have to specify his algorithm for choosing among these algorithms, and you're back with a probabilistic algorithm. Thus there is only one algorithm, thus the prisoner can, in principle, deduce what it is. It is against this fully informed, perfectly reasoning prisoner that the judge makes his strong claim.

But any such probabilistic algorithm has a last day to which a nonzero probability is assigned, and if the judge or his random number generator picks that day, the prisoner will know his fate on the preceding afternoon.

So there is no algorithm with the described properties, and when the prisoner is surprised it only shows that he guessed wrong.

Michael Swaine

Michael Swaine
editor-at-large
mike@swaine.com