



Dr. Dobb's DIGEST

The Art and Business of Software Development March, 2009

Editor's Note by Jonathan Erickson	2
Techno-News The Safe C Library Promoting safer, more secure programming.	4
Features Perils of Going Parallel by Steve Apiki When going parallel, processes and teams—not just the code—must change.	6
Multicore Processors by Richard Kaufmann and Bruce Gayliard How long will the pain last?	8
Porting D to the Mac by Walter Bright Moving the D programming language to a new platform.	11
Crash Course in Lightweight Code Review by Jason Cohen Get results in a week.	14
The Android Developer Experience by Mike Riley Is this the 'droid you've been looking for?	19
Internet Explorer 8 and Compatibility Views by Dino Esposito Beta 2 of Internet Explorer 8 passes the Acid2 test.	26
Columns Book Review by Mike Riley <i>The Art of Performance Testing</i> is a valuable read for any developer and/or tester responsible for ensuring an optimal application execution experience.	27
Conversations by Jonathan Erickson When it comes to parallel computing, performance can be a balancing act.	28
Effective Concurrency by Herb Sutter Illustrating best practices for using threads — techniques that will make concurrent code easier to write correctly and to reason about with confidence.	31

Dr. Dobb's Excellence in Programming Award



By Jonathan Erickson,
Editor In Chief

Past Award Recipients

- **Alexander Stepanov**, for his work on the C++ Standard Template Library.
- **Linus Torvalds**, a name synonymous with Linux.
- **Larry Wall**, author of Perl.
- **James Gosling**, chief architect of Java.
- Ronald Rivest, educator, author, and cryptographer.
- **Gary Kildall**, for his work in operating systems, programming languages, and user interfaces.
- **Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson**, authors of *Design Patterns: Elements of Reusable Object-Oriented Software*.
- **Guido van Rossum**, Python's creator.
- **Donald Becker**, Linux networking contributor and chief investigator of the Beowulf Project.
- **Jon Bentley**, computer science author and researcher.
- **Anders Hejlsberg**, developer of Turbo Pascal and architect of C# and the .NET Framework.
- **Adele Goldberg and Dan Ingalls**, pioneers in Smalltalk and object-oriented programming.
- **Don Chamberlin**, a database researcher and coauthor of SQL.
- **P.J. Plauger**, a longtime champion of the C/C++ programming languages.
- **Guy L. Steele**, for a broad range of contributions to software development.
- **Grady Booch**, for his contributions to object-oriented programming, software architecture, and modeling.
- **Bruce Schneier**, for his contributions to computer security.
- **Bjame Stroustrup**, for creating and supporting the C++ programming language.

Dr. Dobb's Excellence in Programming Award is annually bestowed on individuals who, in the spirit of innovation and cooperation, have made significant contributions to the advancement of software development. The first award was given out in 1995, recognizing Alexander Stepanov, for his work on the C++ Standard Template Library, and Linus Torvalds, for creating Linux.

The recipient of the 2009 award is one of the world's foremost experts on C++ software development — Scott Meyers. In addition to providing training and consulting services to clients worldwide, Meyers is author of the best-selling Effective C++ book series — *Effective C++*, *More Effective C++*, and *Effective STL*), wrote and designed the innovative Effective C++ CD, is consulting editor for Addison-Wesley's Effective Software Development Series, and was a founding member of the Advisory Board for *The C++ Source*. He's written for numerous magazines, including *Dr. Dobb's Journal*, the *C/C++ Users Journal*, and *IEEE Software*.

He has also served as a technical advisor to several start-up companies. A programmer since 1972, Meyers holds an M.S. in Computer Science from Stanford University and a Ph.D. from Brown University.

Join us in congratulating Scott Myers for being this year's recipient, and thanking him for his many contributions to the art and science of software development.

Also join us in congratulating Barbara Liskov for being the 2009 recipient of the Association for Computing Machinery's A.M. Turing Award, one of the highest honors in science and engineering, for her pioneering work in the design of computer programming languages. Liskov's achievements underpin virtually every modern computing-related convenience in people's daily lives.

Liskov, the first U.S. woman to earn a PhD in computer science, was recognized for helping make software more reliable, consistent and resistant to errors and hacking. She is only the second woman to receive the honor, which carries a \$250,000 purse and is often described as the "Nobel Prize in computing."

Liskov heads the Programming Methodology Group in the Computer Science and Artificial Intelligence Laboratory at MIT, where she has conducted research since 1972. Last year, she was named an Institute Professor, the highest honor awarded to an MIT faculty member.

Liskov's early innovations in software design have been the basis of every important programming language since 1975, including Ada, C++, Java and C#.

Liskov's most significant impact stems from her influential contributions to the use of data abstraction, a valuable method for organizing complex programs. She was a leader in demonstrating how data abstraction could be used to make software easier to construct, modify and maintain. Many of these ideas were derived from her experience at MIT in building the VENUS operating system, a small timesharing system that dramatically lowers the cost of providing computing and makes it more interactive.

In another contribution, Liskov designed CLU, an object-oriented programming language incorporating clusters to provide coherent, systematic handling of abstract data types. She and her colleagues at MIT subsequently developed efficient CLU compiler implementations on several different machines, an important step in demonstrating the practicality of her ideas. Data abstraction is now a generally accepted fundamental method of software engineering that focuses on data rather than processes.



Scott Meyers



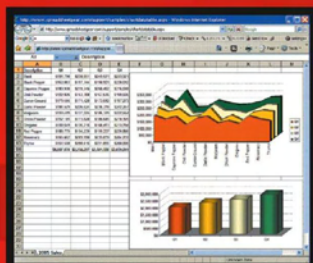
Barbara Liskov

A stylized, handwritten signature in blue ink that reads "Jonathan Erickson".

Microsoft Chose SpreadsheetGear...

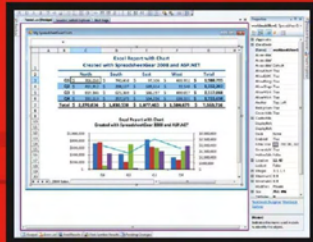
"After carefully evaluating SpreadsheetGear, Excel Services, and other 3rd party options, we ultimately chose SpreadsheetGear for .NET because it is the best fit for MSN Money."

Chris Donohue, MSN Money Program Manager.



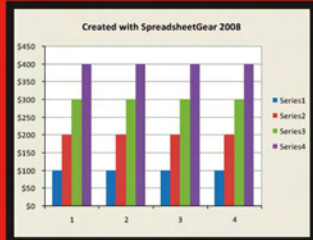
ASP.NET Excel Reporting

Easily create richly formatted Excel reports without Excel using the new generation of spreadsheet technology built from the ground up for scalability and reliability.



Excel Compatible Windows Forms Control

Add powerful Excel compatible viewing, editing, formatting, calculating, charting and printing to your Windows Forms applications with the easy to use WorkbookView control.



Create Dashboards from Excel Charts and Ranges

You and your users can design dashboards, reports, charts, and models in Excel rather than hard to learn developer tools and you can easily deploy them with one line of code.

Download the FREE fully functional 30-Day evaluation of SpreadsheetGear for .NET today at **www.SpreadsheetGear.com**.



The Safe C Library

Promoting safer, more secure programming

The recently released Safe C Library implements a subset of the functions defined in the ISO TR24731 specification, which is designed to provide alternative functions for the C Library (as defined in ISO/IEC 9899:1999) that promotes safer, more secure programming in C.

The Safe C Library (available for download here) provides bound checking memory and string functions per ISO/IEC TR24731. These functions are alternative functions to the existing Standard C Library. The functions verify that output buffers are large enough for the intended result, and return a failure indicator if they are not. Optionally, failing functions call a "runtime-constraint handler" to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are reentrant: they never return pointers to static objects owned by the function.

Included in the Safe C Library are extensions to the specification to provide a complementary set of functions with like behavior. The library can be used to mitigate (that is, lessen) software security problems. When used properly, these functions decrease the danger of buffer overruns and the vulnerability of cyber attacks.

The ISO TR24731 Bounds Checking Interface documents indicate that the key motivation for the new specification is to help mitigate the ever-increasing security attacks — specifically the buffer overrun. According to the rationale document:

Buffer overrun attacks continue to be a security problem. Roughly 10% of vulnerability reports cataloged by CERT from 01/01/2005 to 07/01/2005 involved buffer overflows. Preventing buffer over-

runs is the primary, but not the only, motivation for this technical report.

The rationale document goes on to say that:

...these only mitigate, that is lessen, security problems. When used properly, these functions decrease the danger buffer overrun attacks. Source code may remain vulnerable due to other bugs and security issues. The highest level of security is achieved by building in layers of security utilizing multiple strategies.

The rationale document also lists the following key points for TR24731:

- Guard against overflowing a buffer
- Do not produce unterminated strings
- Do not unexpectedly truncate strings
- Provide a library useful to existing code
- Preserve the null terminated string datatype
- Only require local edits to programs
- Library-based solution
- Support compile-time checking
- Make failures obvious
- Zero buffers, null strings
- Runtime-constraint handler mechanism
- Support reentrant code
- Consistent naming scheme
- Have a uniform pattern for the function parameters and return type
- Deference to existing technology

The Safe C Library only implements a subset of the functions defined in the specification. Included in the library are extensions to the specification to provide a complementary set of functions with like behavior. Among the design considerations for Safe C are:

- Use of *errno*. The TR24731 specification says an implementation may set *errno* for

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

EDITORIAL
MANAGING EDITOR
Deirdre Blake
COPY EDITOR
Amy Stephens
CONTRIBUTING EDITORS
Mike Riley, Herb Sutter
WEBMASTER
Sean Coady

AUDIENCE DEVELOPMENT
DIRECTOR OF AUDIENCE SERVICES
Kristi Cunningham
AUDIENCE SERVICES MANAGER
Gwen Clark

CLIENT SERVICES MANAGER
Gordon Peery
SERVICES MARKETING
COORDINATOR
Laura Robison

DR. DOBB'S

600 Harrison Street, 6th Floor, San Francisco, CA, 94107. 415-947-6000.
www.ddj.com

UBM Limited

Kevin Prinz Chief Information Officer
Pat Nohilly Senior Vice President,
Strategic Development and Business Administration
Anne Marie Miller Corporate Senior Vice President, Sales
Marie Myers Senior Vice President, Manufacturing
Alexandra Raine Senior Vice President, Communications

TechWeb

Tony L. Uphoff Chief Executive Officer
Bob Evans Senior Vice President and Content Director
Eric Faurot Senior Vice President, Live Events Network
Joseph Braue Senior Vice President, Light Reading Communications Network
John Siefert Vice President and Publisher, InformationWeek Business Technology Network
Scott Vaughan Vice President, Marketing Services
Greg Kerwin Vice President, Global Development
John Ecke Vice President, Financial Technology Network
Jill Thiry Publishing Director
John Dennehy Vice President, Finance
Fritz Nelson Executive Producer, TechWeb TV
Scott Popowitz Senior Group Director, Audience Development
Beth Rivera Vice President, Human Resources

techweb™

the functions defined in the technical report, but is not required to. This library does not set *errno*. The library does use *errno* return codes as required by functional APIs. Specific Safe C String and Safe C Memory *errno* codes are defined in the *safe_errno.h* file.

- Runtime-constraints. Per the spec, the library verifies that the calling program does not violate the function's runtime-constraints. If a runtime-constraint is violated, the library calls the currently registered runtime-constraint handler. Per the spec,

Optional features and system-dependent features can be configured by macros in the Safe C Library header files

multiple runtime-constraint violations in the same call to a library function result in only one call to the runtime-constraint handler. The first violation encountered invokes the runtime-constraint handler. The runtime-constraint handler might not return. If the handler does return, the library function whose runtime-constraint was violated returns an indication of failure as given by the function's return.

- *rsize_t*. The specification defines a new type. This type, *rsize_t*, is conditionally defined in the *safe_lib.h* header file.
- *RSIZE_MAX* Macro. The specification defines the macro *RSIZE_MAX*, which expands to a value of type *rsize_t*. The specification uses *RSIZE_MAX* for both the string functions and the memory functions. This implementation defines two macros: *RSIZE_MAX_STR* and *RSIZE_MAX_MEM*. *RSIZE_MAX_STR* defines the range limit for the safe string functions. *RSIZE_MAX_MEM* defines the range limit for the safe memory functions. The point is that string limits can and should be different from memory limits.
- Configurations. Optional features and system-dependent features can be configured by macros in the Safe C Library header files. These can be found by a grep of the files: `grep CONFIGURE *.h`.
- Header files. The specification states the various functions would be added to existing Standard C header files: *stdio.h*, *string.h*, etc. This implementation separates the memory-related functions into the *safe_mem_lib.h* header and the string-related functions into the *safe_str_lib.h* header. The make file builds a single library *safe_lib.a* in the *./lib* directory. It is possible to split the make such that a separate *safe_mem_lib.a* and *safe_str_lib.a* are built. It is also possible to integrate the prototypes into the Standard C header files, but that may require changes to your development toolchain.

Acknowledgments

Thanks to Safe C Project Administrator Bo Berry, who prepared the readme file on which this short description is based.

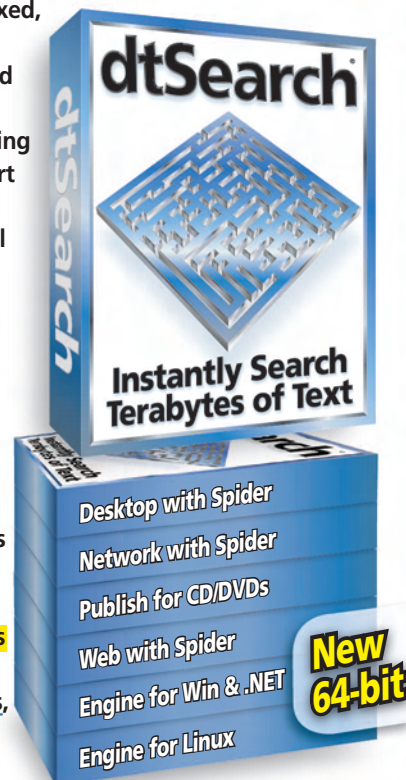
Instantly Search Terabytes of Text

- ◆ dozens of indexed, unindexed, fielded data and full-text search options (including Unicode support for hundreds of international languages)

- ◆ file parsers / converters for **hit-highlighted** display of all popular file types

- ◆ Spider supports static and dynamic web data; **highlights hits** while displaying **links**, formatting and **images** intact

- ◆ API supports .NET, C++, Java, databases, etc. New .NET Spider API



The Smart Choice for Text Retrieval® since 1991

- ◆ "Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second" – *InfoWorld*
- ◆ "For combing through large amounts of data," dtSearch "leads the market" – *Network Computing*
- ◆ dtSearch "covers all data sources ... powerful Web-based engines" – *eWEEK*
- ◆ dtSearch "searches at blazing speeds" – *Computer Reseller News Test Center*

See www.dtsearch.com for hundreds more reviews, and hundreds of developer case studies

Contact dtSearch for
fully-functional evaluations

1-800-IT-FINDS • www.dtsearch.com

Perils of Going Parallel

Processes and teams, not just the code, must change

by Steve Apiki

As multicore servers become commonplace, IT organizations are faced with the task of getting the most bang from their server buck. Particularly for applications developed in-house, the decision to reprogram for parallel execution goes well beyond using some new programming techniques. Going parallel requires revising tried-and-true processes and rethinking how project teams are built.

For those involved in staffing and assembling project teams, managing multicore development may involve finding and allocating people with thread programming expertise, and they're hard to find. Along with that, deploying highly threaded applications introduces systems concerns. All of these problems can be handled, but because the solutions may require crossing departmental lines, they'll likely require high-level management involvement.

Up-Front Decisions

The most fundamental choice in creating multicore-aware applications involves parallel abstraction. Parallelism may come from a set of loosely coupled services, or it might come from threads in a single-process system. In between these extremes, other parallel models may apply. Although the choice of parallel abstraction is a software design decision, it becomes a strategic business consideration because everything from staffing to server configuration rides on where the line is drawn.

Low-level threading is hard—really hard—to get right, which is why it makes sense to consider higher level abstractions. Moreover, some parallel models already have proved successful in mainstream applications—Web services communicating through SOAP or REST, cooperating applications

such as application and database servers running on multiple virtual machines, multiple independent processes running on a single machine and communicating through pipes.

While each of these approaches sidesteps the complexity of threading, the trade-off is scalability. The most scalable designs are those that apply threading to data parallelism. Task-parallel applications scale less well, and there's more overhead in applications composed of processes than in those built from lightweight threads. In general, the finer-grained the abstraction, the more scalable the application, and the trickier the coding.

Large projects probably will include several modes of parallelism. In service-based applications, for example, selected services may be internally threaded. These may be services with high compute loads, or that include some natural data parallelism.

Managing Threading Expertise

Building a team to do threaded software development poses a unique challenge for development managers. First, threading expertise is hard to come by. Second, the people with domain or legacy application knowledge are unlikely to also have deep threading experience. Building a successful team means finding or cultivating the right mix of expertise.

Teams should include someone strong in parallel algorithm design. Parallelism should be thought of as a structural aspect of application design, so it's important to have strong parallel development skills in an architectural role.

Along with the architect, you need application experts who can learn threading techniques. Testers also should be trained for multithreading. In general, the team should be structured with

strong parallel skills in a lead role, but with application and domain experience counting for more among other team members. A reasonable goal is a team with 80 percent of its expertise in the application domain and 20 percent in threading and parallelism.

In migrating legacy code, a developer familiar with the project—ideally, the developer who originally wrote the code—should make the changes to introduce threading. Threading experts can provide guidance, but code familiarity is more important when determining how a module should be restructured.

All teams, especially those in which threading experience is light, should consider threading libraries for development. Threading libraries, such as Intel's Threading Building Blocks or OpenMP, can ease the transition by delivering thread-level scalability and performance while providing developers with a more abstract programming model. And libraries aren't just a transitional tool; by insulating developers from thread-scheduling considerations and the details of working with individual cores, they also provide a way to move up to processors with more cores without changes to the application.

Similarly, teams can take advantage of thread pool mechanisms built into development environments. The .NET CLR and Java 5 both offer thread pool abstractions that can provide significant performance improvement.

Changes To The Development Cycle

Moving to multicore means adjustments to every phase of the development cycle, from project planning to application tuning.

Intel, for instance, suggests a four-step general process for parallel development: discover, expression, confidence, and optimization.

In the discover step, parallelism specialists try to find the natural parallelism in the application, including determining the right

level of abstraction. Other tasks in the discovery step include considering refactoring to exploit parallelism and considering tools that may be used for development.

Next, expression involves the parallelism architect and app designers developing a design that uses the parallel modes discovered in the prior step. It's important to work through thread coordination and conflicts in design. This phase includes actual implementation.

The confidence step includes testing the application. Errors mean going back to discovery or expression. Multithreaded development introduces testing requirements for correctness beyond the requirements of single-threaded implementations. For legacy conversions, the multithreaded version also must be tested for consistency with the single-threaded version.

Optimization occurs after the application has met functionality requirements. Here, significant gains can be had by tuning locking, cache interactions, and the like. Low-level thread expertise is critical here.

The development cycle for parallel applications is iterative, and teams may need to return more than once to earlier steps until they hit correctness and performance targets. Performance limitations found during optimization may require returning to the expression step to revise the implementation.

The takeaway here is that parallelism must be considered in each stage of the development cycle. Adding parallelism too late in the process means a bolted-on threading implementation that usually fails to exploit all of the parallelism in the application. Or a design left untuned leaves performance on the table.

In Production

Deploying several heavily threaded applications on a single server introduces its own set of problems. As more applications get threaded, thread scheduling becomes increasingly important to overall system performance. Max Domeika, a senior staff soft-

ware engineer at Intel, says thread scheduling should be considered outside of the scope of most application development projects. He advises developers to focus on parallelism within the application, relying on the operating system or libraries to handle thread scheduling. For a large system with several processes and threads, the system architect and performance engineer should understand the impact of the cooperative processes on the shared hardware. "They will need to verify that their performance constraints can be met," says Domeika.

Power Considerations

Multicore processors promise reduced power consumption in the data center as well as scalable performance for parallel applications. Multicore servers can lower power and cooling costs relative to single-core servers by sharing supporting hardware among cores and by increasing the amount of computing power available at a given clock speed. But here, too, there's a system design consideration that must reconcile the requirements of the data center manager with those of development.

Maximizing performance per watt is a data center goal; minimizing system response time is a development goal. Negotiating between the two means balancing system throughput for optimal power performance and system latency for shortest response time. High-level management must play a hand in managing these goals.

For management, the move from single core to multicore is a deep technological change that requires building effective development teams. Another challenge lies in reaching across IT groups to develop an integrated approach to parallel software development and deployment. This coordinating and involving development, systems, and facilities teams in appropriate roles.

—Steve Apiki is senior developer at Appropriate Solutions, a consulting firm that builds server-based software solutions.

Multicore Processors

How long will the pain last?

by Richard Kaufmann and
Bruce Gayliard

Until a few years ago, the processor hardware community translated Moore's Law of transistor density directly into single-threaded performance gains as a result of increasing clock frequencies. Lately, this translation has been hampered by the effects of clock frequency on power consumption and heat generation. The new reality is that per-thread performance is essentially static, and an increase in performance is delivered by an increase in the number of available processor cores per socket. This greatly simplifies a processor developer's job in two ways: First, it avoids radically increasing per-socket power consumption; and second, it holds the complexity of a single core somewhat static over time. Thus, processor design remains a tractable discipline.

However, nothing in life comes for free. Just as RISC stood for "Relegate the Important Stuff to the Compiler," increasing the number of threads per socket places an enormous stress—to parallelize applications and reap the performance increases—on almost the entire computing industry. Those affected include the system manufacturers, the compiler writers (who are used to being the punching bags), the operating system designers, the application writers, and finally (and inevitably) the users. The single-threaded application typically does not benefit from multicore hardware, it may actually suffer degraded performance; and the great majority of applications on the market today are single-threaded in design. In general, clock frequency increases allowed developers to avoid the inevitable transition to multiprocessor environments (and thus the parallelization of their applications), but this free ride is now over.

In addition to the difficulties of parallelizing an application, there is a limit to the performance gain

that a parallelized version of an application can achieve. This is defined by a variant of "Amdahl's Law" (which fundamentally states that an application cannot run faster than the aggregate of its sequential components). An analysis of the application may indicate that parallelization at the code level will not yield the sought-after performance gains, and that some other approach (for instance, running multiple copies of a program to increase throughput) will be more effective. Furthermore, on the hardware front, increasing core counts will result in pressure on communication bandwidth capacities, particularly for memory accesses. Maintaining the traditional clocked bus architectures will result in a need for increased bus clock frequencies, again pushing on the very power dissipation issues that resulted in the move to multicore architectures in the first place. Increasing the width of the involved busses may provide some temporary relief, but that approach can only be viewed as a patch.

Despite these challenges, the component manufacturers (Intel, AMD, etc.) are gambling that system and software developers will pick up the ball and move toward parallelizing their applications quickly enough to keep up with the core count arms race.

Okay, It's Not All Doom and Gloom

Luckily, some applications are inherently scalable and can accommodate exponential growth in data by applying additional computing resources. For example, digital content creators generally assign one frame of a movie to a single thread. It really doesn't matter how long that thread takes to complete as long as the number of thread completions per unit of time scales appropriately.

Other examples include some High Performance Computing (HPC) applications (often developed using the Message Passing Interface (MPI) or OpenMP programming models) or other applications developed using Google's MapReduce library (see also Apache Hadoop).

Furthermore, a nice property of the transition from clock frequency increases to multicore processors is that memory latency is no longer a worsening problem. Those of us who remember the utter joy of upgrading from 026 to 029 keypunches (the 10 or so of you who are now in danger of dropping your dentures!) also remember when memory systems ran at the same speed as the CPU; a word of memory was one clock tick away from the CPU. As processor speeds increased, this relationship failed to keep pace, up to the point where at 3GHz (.333ns cycle) and approximately 80ns latency, a word of memory is approximately 240 ticks away from the CPU.

A particularly helpful side effect of keeping a constant CPU frequency is that the relative time (in clock ticks) to access memory no longer increases with new generations of processors.

Many applications aren't able to take full advantage of the current CPU clock speeds as their performances are dictated by memory latency (for example, programs that spend most of their time chasing linked lists). If many copies of these programs are run in parallel, latency can be overcome—the cores collectively can provide enough outstanding loads to “hide” the underlying memory system latency. In fact, early multicore systems were designed with such workloads in mind.

Also, keeping a relatively stable clock speed has had a positive affect on CPU power consumption. Calculations by Intel for 65nm CPUs showed that increasing the core frequency by a factor of 1.3 would double CPU power consumption. Doubling the core count and retarding the clock by 10 percent, in contrast, would boost performance by a factor of 1.8 with no increase in CPU power consumption.

The Reality

But what if the application can't be scaled by simply increasing the number of independent runs of identical programs? What if you want to double the speed of your single application every 18 to 24 months in accordance with the rights and privileges afforded by the prior single-threaded performance aspect of Moore's Law? And you want to do this without having to become a wizard at parallelizing your program, or even worse, rewriting your application? If you're busy nodding your head up and down, you'll agree with this statement: Exploiting parallelism within a socket should be the responsibility of the compilers, tools, schedulers, operating systems, and so on. It's okay if parallelism across sockets remains “hard,” but single socket stuff should just happen automatically.

Unfortunately, the industry does not have the technology to accept the responsibility of automatically taking advantage of multicore processors. It is up to the software developer to rewrite an application, and develop an understanding of parallelism in order to do so.

The Solution: A Collaborative Approach

With these issues in mind, there are many industry specialists that are devising technologies to make the parallelization task easier and more productive. However, technology is not enough. The entire industry must work together to address “the multicore menace.” Areas to be addressed include:

- Advancements in programming languages and programming styles from academia and industry.
- Compilers, tools, and runtimes to support these languages and styles. Parallelism across sockets, as well as among cores, will require support (but, as discussed, this may be “hard to do”). Furthermore, support will be required for optimization of memory bus bandwidth utilization.
- Operating systems and system schedulers (e.g. SLURM) that account for multicore topologies, and take compiler- and application-supplied hints to prop-

erly lay out a parallel application on a socket/server/cluster.

- Algorithms that require reconsideration for their applicability to multicore programming. In particular, transactional memory systems incur a high cost for hot elements in a data structure; e.g., tree balancing algorithms that frequently update the root of a tree are inherently less scalable than those that don't.
- Business models that recognize multicore realities. Application licensing costs sometimes scale with thread count or with socket count, and incredibly, sometimes at a fractional power of thread count. The industry must quickly converge on a pricing model that is fair and understandable.
- Education of developers and users concerning the technologies being developed for multicore processors.

This list is a good start, but not all inclusive. It is imperative that the industry distinguishes the pros and cons of each technology, identifies the preferences of the developers and users, and addresses those preferences. The industry must quickly provide effective solutions to parallelize applications, and it is vital that developers and users are involved in defining those solutions.

Along these lines, a number of developments have progressed. Some examples:

- Hewlett-Packard has collected a number of products into a Multicore Toolkit providing developers with “one-stop shopping” for products and information. HP also provides HP-MPI, HP-UPC (a Unified Parallel C implementation), and other products useful for multicore development. Additionally, HP has been involved in improving the open source support; for example, enhancing SLURM to support multicore resources.
- Acumem SlowSpotter samples execution of a live application to obtain information about memory bus and cache utilization characteristics. The data is subsequently analyzed, giving the user direct relationships between memory access patterns and analyzed code. Suggestions are provided, based on the results of an analysis, as to how performance improvements can be achieved.
- Cilk Arts is working on Cilk++, which simplifies the task of parallelizing code. A small set of specialized keywords are introduced into what would otherwise

be a compliant C program. The keywords indicate the functions that can be parallelized and work units that comprise those functions. The runtime system schedules the work units among the available processing elements, using a “work stealing” paradigm.

- Rapidmind represents a different technique for specifying parallelizable code.

each handling some element of the overall application.

- eXludus brings a system resource manager to the table, particularly useful for time-sharing systems. Scheduling of processes is dynamically adjusted based on resource utilization, facilitating optimal use of the overall system and associated throughput improvements.

A fundamental change in the memory subsystem will eventually

be required to keep up with core counts

An application writer specifies a parallelizable kernel function of an algorithm. This kernel is then applied over a set of data, e.g. an array calculation. The runtime system then relies on a Just In Time (JIT) compiler to build the parallelizable code, taking into account the available processing elements; for example, a multicore CPU, a graphics processor, or perhaps some other exotic ASIC. If this technique maps well to an algorithm, it can not only distribute the work over available CPU cores in a transparent manner, it can also allow the use of a graphics card as an application accelerator.

- Intel Ct takes a somewhat similar approach, but is more explicitly vectorized. Using a C++ template to describe a Throughput VECtor (TVEC), the programmer defines vectorized operations. The compiler and runtime system divides the work into units and then distributes the work units across the available processing elements. Intel has also announced plans for the Intel Parallel Studio, a set of resources targeted at parallel program development to be integrated with Microsoft's Visual Studio.
- Microsoft, in collaboration with Intel, is working on the Task Parallel Library (TPL) to allow expression of parallelism with normal function calls. The resultant code will automatically utilize the available cores. TPL is another implementation that relies on “work-stealing” to dynamically distribute the load among the available threads.
- Rogue Wave, attacking the problem from another angle, provides software that supports a Service Oriented Architecture. Rather than parallelizing within a program, the developer can use Hydra to create a service grid and execute a number of concurrent processes,

- Visual Numerics provides enhanced mathematics libraries that allow for parallelized operation within the library calls, supporting OpenMP and MPI. The level of parallelism can thus be adjusted using environment variables.
- Transactional memory is an idea that has become a popular topic in academic and industry circles. TM improves on traditional mutual exclusion locks by removing the cost of uncontested locks while greatly increasing the cost of conflicts. TM also has some nice properties that make things easier for large applications comprised of independent components.
- The DARPA High Productivity Computing System (HPCS) effort has yielded three proposed language designs: Sun's Fortress, IBM's X10, and Cray's Chapel. These languages go beyond scaling within a single socket, and are intended to allow programmers to be more productive when programming exascale systems (comprising thousands of sockets).

What's Next?

The memory latency aspect mentioned earlier may work to the advantage of the multicore designers for a while, because an increase in the number of cores will come with an increase in memory bus bandwidth requirements. A fundamental change in the memory subsystem will eventually be required to keep up with core counts. One such technology might be stacked memory—a technique that places memory on a die directly above or below the CPU. Each core would have access to the memory directly attached to it, and bandwidth could scale with core counts. Another possible choice is

to replace the copper connections to memory with optical links. A move away from the traditional globally clocked bus architecture is also possible, replacing it with, for example, a self-timed communications fabric. A melding of techniques may be necessary to give the right combination of price, power consumption, capacity, and bandwidth.

Although it makes sense in 2008 to assume a modest number of sockets will have access to a coherent shared memory, there may come a time when this is no longer true. For example, it may be possible that even cores within a socket may be partitioned into multiple coherency domains. This is game changing: Applications written today to be “multicore aware” quite commonly assume coherent memory access across all cores. When this is no longer true, other techniques (such as message passing) are required to take advantage of all the cores. Thus, even single socket applications might look more like cluster-aware distributed memory codes currently popular in the supercomputing community.

Conclusion

As stated, though multicore techniques have made processor design more practical, they have complicated the lives of application developers, tool writers, and so on. Therefore, the entire computer ecosystem (academic and commercial) must respond to this challenge, and quickly develop a set of technologies to allow developers and users to take advantage of the potential performance improvements afforded by new platforms. Though this is well underway, the pain may last for years while application developers gain an understanding of how to use these emerging technologies. They must adjust to the relatively constant clock rates of newly developed processors and take advantage of the explosion of available computing resources in the form of cores. The hardware is way ahead of everyone else at this point, and catching up will be very, very difficult.

—Richard Kaufmann is a Distinguished Technologist at Hewlett-Packard. Bruce Gayliard is a High Performance Computing Software Engineer at Hewlett-Packard.

Porting D to the Mac

Moving to a new platform

by Walter Bright

D is a systems programming language. Its focus is on combining the power and high performance of C and C++ with the programmer productivity of modern languages like Ruby and Python. Special attention is given to the needs of quality assurance, documentation, management, portability, and reliability.

The D language is statically typed and compiles directly to machine code. It's multiparadigm, supporting many programming styles: imperative, object oriented, and metaprogramming. It's a member of the C syntax family, and its appearance is very similar to that of C++. For a quick comparison of the features, see this comparison of D with C, C++, C#, and Java.

D is not governed by a corporate agenda or any overarching theory of programming. The needs and contributions of the D programming community form the direction it goes.

Multiple compilers exist for the D programming language, including the Digital Mars compiler (dmd), the gnu compiler (gdc), the upcoming ldc compiler, and even a .NET D compiler is in the works. They are all based on the same open-source frontend code.

dmd has an X86 code generator. It currently targets Windows and Linux. But now that the Mac is on X86 machines, it opens the possibility for a straightforward implementation of dmd on the Mac. The only way to find out how feasible it is to get a Mac and get to work on it.

Over Christmas, I ordered a MacMini from Amazon. Compiler development doesn't require a powerful machine or lots of memory and/or disk space, so a low-end machine will do nicely. I've never used a Mac before, so I was also curious how the machine would work. It didn't take long to get it up and running. I keep spare keyboards and mice around (because they break often), and plugged them in along with an old monitor. The monitor and mouse worked perfectly, but the Mac had some trouble with the keyboard, and needed to be configured for it. The machine comes with the gnu dev tools, though they needed to be installed separately. I then figured out how to remotely connect to the Mac over the LAN, and (the Mac people will hate me for this) put the Mac in the basement and operates it remotely with a text window.

From a text window, the Mac is just another UNIX machine. I put all the source code to the compiler on it and set about trying to compile it. Most of that consisted of finding all the conditional compilation:

```
#if linux
```

and changing it to:

```
#if linux || __APPLE__
```

Remarkably, there were very few API differences that needed to be accounted for, a couple gcc differences, and the compiler was running. I thought I'd start by assuming that the C ABI on the Mac was the same as for Linux, and configured dmd appropriately.

The first big problem is that dmd can generate both Intel OMF and ELF object file formats, but the Mac used the Mach-O format. The best way to learn a file format is to write a dumper for it (called "dumpobj" for object files). Object file specs are usually wrong in some detail or other, and the Mach-O spec is no exception. The Mac comes with an object file dumper called "otool." Unfortunately, otool doesn't give a clue as to the structure of the object file, nor will it disassemble any code that is not in a __text segment. So I also got the Digital Mars disassembler, obj2asm, converted to work with Mach-O files.

Fortunately, the Mac uses Dwarf for its symbolic debug information format, and dmd has a Dwarf generator, so that should be good to go. But when I first looked at the debug output of gcc, there was a "macinfo" section. Uh-oh, some undocumented Macintosh enhancement. Googling (how indispensable Google has become) "macinfo" revealed my mistake—I had forgotten that Dwarf had a special section for information on C preprocessing macros called "macinfo". I forgot about it because D doesn't have a text macro preprocessor.

Object files on the Mac are all generated as pic (Position Independent Code), necessary so that shared libraries will work. On Linux, pic is an option. Pic is done completely differently on the Mac, so this was where most of the work was so far.

Some other differences:

- Names get an '_' prepended to them, although when using gdb you have to leave off the '_'.
- There's no thread local storage mechanism in the object file format. This is a serious shortcoming, and I'll have to figure out a solution.
- There are special sections for C strings and read-only literals that the linker can compress redundancy out of.

I finally got to the point where dmd would compile "hello world," and using dumpobj to compare object files with that produced by gcc for the C "hello world", it

looked like it should work. The gcc one worked fine, and the dmd-generated one crashed with a segmentation fault. I was really pulling what few strands of hair I had left out over that one, as I could not find anything wrong with the fixups or object layout.

Eventually, I noticed that the gcc version was putting some unneeded space on the stack, and I suspected something was up. Put out the same extra stack, and it started working. Googling around some more, I discovered that code on the Mac that calls any library functions must align the stack to 16 bytes. Tweaking the code generator to do this, now I had “hello world” working in D on the Mac.

It Works—Sort Of

I had “hello world” working, but I was using D as a C compiler to do it. All that existed was a *main()* function. There was a long way to go — functions, data, the runtime library, the test suite, etc. First I'd try and build the runtime library.

Doing so exposed a lot of conditional compilation issues that had no case for OS X. I found that Linux has a bunch of API functions that are missing in OS X, like *getline* and *getdelim*, so some of the library functionality had to revert to more generic code for OS X. I had to be careful, because although many system macros had the same functionality and spelling, they had different expansions. Getting these wrong would cause some mysterious behavior, indeed.

After compiling the library modules, they had to be combined into a library file. Normally this is just done with *ar*, the standard “archive” program, but dmd has a nice feature where it can build the library directly without having to write object files out to disk. Not only is this very fast, but it makes the build process simpler by obviating the need to manage the object files. The archive file format is supposed to be a standard, but of course everyone implements it very differently. The OS X archive format spec is only about half there, and what is there is wrong on a few details. This is where building a file dumper again comes in real handy. By varying the input to *ar*, dumping the resulting archive, and seeing how things change, one eventually figures out the format. If you get it wrong, the OS X linker *ld* likes to help you out with a convenient “Bus error” message. To figure those out, it's back to puzzling out the hex dumps of the archive dmd generated against the one *ar* generated. I did finally figure it out. The archive format has some definite oddities; for instance, entries are aligned on 4-byte boundaries but the object file within each entry must be aligned on 8-byte boundaries, offset by 4. Weird. Anyhow, once that was working and the linker accepted the archive, it was very pleasing to see how fast dmd would chew through the code and build the library.

Unfortunately, none of the library code ran, and I mean none of it. When the compiler was originally designed, I attempted to do the right thing and abstract away the memory model and the object file format. Because I had only one example, the abstraction lines turned out to be in all the wrong places. I coined the term “premature abstraction,” which, like the related term “premature optimization,” is coming up with all the wrong abstractions because you don't know what you're doing. (Andrei Alexandrescu suggested I write a column

about that, but a quick Google search showed I was not original and the term is already in use!)

For example, the data structures being written out were divided into mutable and immutable sections, the immutable stuff being destined for a read-only hardware protected memory section. But on OS X, read-only data sections get marked as “TEXT” and put in with the code, where they are expected to be position independent. Position independence means no relocations are possible at program load time, meaning the read-only data cannot have any pointers in it. If there was a relocation in the TEXT section, the loader didn't complain, you just wound up with garbage for a pointer. So, all the data structure generating code had to be carefully gone through and separated into “has pointers” and “no pointers” blocks, and the “has pointers” stuff gets routed to the regular mutable data segment.

Another tricky problem with position-independent code on OS X is the way data variables are accessed. If they are defined within the module they are referenced in, one indirection is generated. If defined externally, two are necessary. The trouble is, the D programming language allows forward references to data, so there's a chicken-and-egg problem in deciding what to resolve first.

The last problem is D needs to generate some tables that are coalesced with other tables, such as the exception-handling data. To group stuff together, it is put in a specially named segment. The problem is locating the beginning and end of that segment at runtime. The solution is a trick I learned long ago in the DOS world—to always put out that special segment as a trio, always in the same order. The linker thankfully will then maintain the same order in the executable, so to find the beginning of the second segment, a global variable is put in the first and third segments, and the addresses of those globals neatly bracket the table in the second segment. This is the way I'm going to try and do thread local storage, too.

Now that dmd is generating three very different object file formats, where the abstraction lines should be is a lot clearer.

Finally, I got the runtime to start up, run, and shut down successfully. Next, I'll try to get the test suite to run.

It Works!

The short version—it's done and out!

I had thought retargeting the D programming language to the Mac was just an object file format change — a week, maybe two. It turned out to be six weeks. I feel like Yosemite Sam walking through a mine field, managing to step on every single mine in it.

So let's pick up where I left off: I had just gotten the library to compile and a couple sample pieces of code to link to it and run. Now it was time to run the test suite.

It's essentially impossible to develop a compiler without some sort of test suite. The one I use is an ad-hoc collection of every fixed bug and other assortments of things. The beauty of putting in every fixed bug is that the bugs stay fixed, and the quality of the compiler steadily ratchets forward. Over time, this makes for a formidable test suite. If it passes the suite, I know it's at least as good as the previous version, and if it shows to not be, that failing gets added to the suite

for next time. (I've discovered that just throwing volumes of code at the compiler is fairly useless as a test suite. The test code has to be crafted to test specific things and verify correct results.)

The first thing that failed was exception handling. But surprisingly, it only took me about an hour to get that to work. Exception handling is complicated and hard to understand, and I expected a tough slog. The EH design in the backend dates back to when it was a 16-bit compiler (Digital Mars C++ was the only C++ compiler to ever implement exception handling on DOS). The support code moved to 32-bit DOS extenders, then Linux, and now OS X with very little change. The OS X support needed to tweak the assembler bits to keep the stack 16-byte aligned. (Exception handling for Win32 is completely different, using Microsoft's Structured Exception Handling scheme.)

The dmd exception-handling system is completely independent of g++'s. The two do not interact with each other. D is binary compatible with C++ name mangling and single inheritance, and g++ on OS X uses the same protocol as on Linux, which dmd is already compatible with, so that was easy.

My next problem was the floating point failed miserably. Some investigation showed that, uniquely, OS X aligns the CPU's 10-byte reals on 16-byte boundaries. This is 6 bytes of pad for each. I don't know the reason for this; Linux uses 12 and Win32 uses 10. It's just that if you have large arrays of reals, it's going to chew up 60 percent more space. Oh well, it was easy to account for in the code generation.

The worst problem I had was my own fault. The front end is only few years old. But the backend code generator is about 25 years old (it may be the oldest code generator still in professional use!). Although it is well debugged (sporting a thorough test suite), fast, and generates great code, it uses a lot of global variables to communicate. Problem after problem was traced back to the use of global variables. Over time I'd eliminated a lot of them, but there's a lot left. It's hard to change how a function works if there's a back channel of globals

passing state around. Globals break encapsulation, making code difficult to understand.

Of course I wouldn't write it that way today, hopefully I've learned something in the last 25 years. You might ask "Why not just rewrite it?" and certainly that thought comes to mind. The problem is a code generator is about a zillion special cases, most of which interact with each other. Getting that all adjusted, tweaked, and working right across the broad spectrum of code that it must generate is years of work. It's not something you throw away and rewrite lightly. But there is some hope. The Mach-O generating part is a lot nicer than the Elf generating part, which itself is much nicer than the very old OMF generator. And if I'm doing open heart surgery on a particular section, I'll refactor it and rely on the test suite to make sure the patient recovers.

Acknowledgments

Thanks to Sean Kelly for his invaluable help with the more complex OS X system library work. Thanks to Jason House, Andrei Alexandrescu, Bartosz Milewski, Sean Kelly, and Cristian Vlasceanu for reviewing this article.

Epilogue

I've learned a few more things now that the Mac OSX version of dmd is out.

The first thing that happened was dmd failed on version 10.4 of OSX with a lovely "bus error" message. dmd was developed and tested on version 10.5. On Windows I was used to things working across the line of Windows versions, so this was a bit of a surprise. Various instructions for fixing it conflicted with each other, and I didn't have a 10.4 system to test it on, so this involved sending each build to people who did until I got it right.

That got the compiler executables working on 10.4. But a more serious problem developed when trying to get the runtime library working on 10.4. The D programming language is designed to work with multiple threads, and the pthreads implementation on 10.4 is weak. The tradeoff is

spending time trying to accommodate it, versus 10.4 being officially obsoleted when Apple updates OSX in the next year. We decided it probably wasn't worth the effort to support 10.4.

Daveed Vandevorde emailed me that the reason long doubles are 16 bytes long on the x86 Mac is so the struct layouts are binary compatible with those for the PowerPC Mac. This makes perfect sense. I was concerned that I was missing some efficiency reason.

The accuracy of some of the standard math library functions dropped from 19 decimal digits to 16. Those functions were implemented by simply calling the corresponding ones in the C standard library. I suspect the problem is that, in the implementation of the C versions, there is a conversion to double precision at some point which would completely account for the loss in accuracy. This provided the motivation to replace those functions with our own. Don Clugston implemented them to produce the full 19 digits of accuracy, and now the math functions reliably produce the same results across x86 platforms. One nice thing about the Mac coming only recently to the x86 platform is the math functions can assume modern CPU hardware, and so they use the newer and faster hardware instructions.

One nice lesson learned is that the test suite we have for dmd works. Once it passed the test suite, few OSX specific problems have cropped up in the wild.

A further change since the initial OSX release is that the release now includes the full dmd compiler source code, so you can build it and see how it works yourself. If you're even more interested in how compilers work, I'm teaching a Compiler Construction seminar in the fall.

Thanks to Sean Kelly and Andrei Alexandrescu for reviewing a draft of this.

—Walter Bright graduated from Caltech in 1979 with a degree in mechanical engineering. He worked for Boeing for 3 years on the development of the 757 stabilizer trim system. He then switched to writing software, in particular compilers, and has been writing them ever since.

Crash Course in Lightweight Code Review

Get results in a week

by Jason Cohen

Most programmers agree that another pair of eyes on your code will uncover bugs and disseminate knowledge across development teams. But many also recognize that peer review can waste a lot of time.

So how do you get started in such a way that you 1) don't waste time, 2) match the process to your team and your goals, and 3) have a clear way to evaluate results? So many code review techniques exist, and each with pros and cons, so which are right for your team? Even if you're unwilling to spend the time to review all your code, perhaps spending a little time reviewing a specific subset would be worthwhile.

The only way to know is to try it for yourself. Use these tips to simplify, expedite, and measure the process.

What Is Code Review and How Do You Do It?

First, let's make sure we're on the same page. What is "code review?" The answer varies depending on whom you ask, and when; code review has changed dramatically in recent years.

Thirty years ago, if you wanted a well-known and measurable process that delivered results, your only choice was the dreaded Formal Inspection—a seven-phase, multi-meeting, paperwork-heavy system popularized by Michael Fagan at IBM. Properly done, it takes over 10 person-hours to review at most 200 lines of code—a sluggish rate that makes this system impractical (not to mention excruciating) even if it does uncover bugs.

In the past 10 years a variety of evidence [1][2][3][4] has shown that many of the components of the traditional "heavyweight" inspection process are inefficient or even unnecessary.

"Lightweight" modern code review methods include:

- **Over-the-Shoulder:** The reviewer physically looks over the author's shoulder and is walked through the code.

Pros: Easy to implement, no tools required.

Cons: Interrupts reviewers, hard to know which code is reviewed, impossible to measure and track. Requires coordinating with someone else's schedule.

- **E-mail Pass-Around:** Code differences are e-mailed automatically by the version-control system. Reviewers respond to changes when they have time.

Pros: Easy to set up, nothing to purchase, works remotely.

Cons: No tracking—don't know if found defects were fixed, often reviews never happen because everyone assumes someone else looked at it, no metrics; reviews "fizzle out" instead of finishing. Hard to follow conversations as e-mails get replied to and forwarded among several people.

- **Tool-Based Review:** Use a development tool specifically designed to assist in code review. A variety of open-source and commercial tools exist.

Pros: Removes most of the busywork associated with file-collection, discussion/defect management, and metrics/reporting. Tracks reviews and defects.

Cons: Have to learn and integrate another tool, can cost money.

- **Pair-Programming:** Two developers at the same keyboard and monitor working on the code together.

Pros: Deep level of review; teams know review is happening but it's difficult to track.

Cons: Large time investment, some developers dislike it. Requires coordinating with someone else's schedule.

Pick a method that most closely matches the existing culture of the team. Use the pros, cons, and amount of time you have to select one and stick with it for one week before you start to make changes.

If you're only going to do a little bit of review, what should you review to have the greatest impact?

Getting Started: Baby Steps

It's difficult to introduce a process that involves a large disruption: projects get behind, people rebel, and the process fails—regardless of potential future efficiency.

It's even harder when there's push-back from developers who aren't yet convinced the process makes sense. This effect is common in code review because people recall previous bad experiences with time-sucking formal inspections or with review processes that were designed with more rules than thought. There's also the worry of hurting people's feelings.

So the best way to introduce code review is at a small scale, with only a handful of files, incrementally, and with full transparency to the team. Taking baby steps is especially effective against those who say, "We cannot change our process right now."

Given this, you'll want to select a style of code review that is lightweight and easy yet provides transparency by tracking things like "number of defects found" and "time spent in review." Any of the methods listed above will work; some are easier to get going but require more manual work to collect metrics. A good compromise as you're getting started is to use a code review tool for free, either by trialing a commercial tool or using an open-source tool. That way you get

the benefit of structured guidance through the review but you can do it without dipping into the budget.

Low-Hanging Fruit: Review What Matters Most

Besides the social and training benefits of introducing a small change to your process at a time, you may find that you can reap significant benefits even if you never get to the point where you're reviewing every line of code. If it turns out to be valuable only in specific circumstances, that's okay. Not every process needs to cover every file in every project over all time and all developers.

So if you're only going to do a little bit of review, what should you review to have the greatest impact?

Here are some ideas:

- Review changes to the stable branch
- Review changes to the core module—the one all other code depends on
- Review changes to the "Top 10 Scariest Files" as voted by the developers
- Review only unit tests
- Review only incremental changes, not entire files
- Review code whenever a developer feels it's necessary

Once you get comfortable you can cast your net to a wider set of code if it makes sense.

By then you'll have a good idea of what you're getting from review and how much time it's taking, so you'll have the information you need to decide exactly how to expand the process.

Not All Bugs Are Created Equal

It's worth mentioning that not all bugs are easy to find in code review. GUI problems are a great example. GUI code is often generated automatically and it's almost impossible to read the code and know if the GUI makes sense. In QA, you can instantly spot visual flaws, and mechanical problems are usually straightforward as well. Similarly, fixing bugs in GUIs is usually a simple, fast matter.

Clearly, the type of bug found in code review matters to your efficiency and effec-

tiveness calculations. Track the "category" of bug so you can determine efficiency by type. Example types are:

- Algorithm
- Build
- Data Access
- Documentation
- Error-Handling
- Interface
- Maintainability
- Performance
- Robustness
- Style
- Testing
- User Interface

This way you can empirically determine where your reviews are efficient and where they're a waste of time.

Measure Success

Code review takes up valuable developer hours, so it'd better be worth it!

Researchers have proposed many ways of measuring the productiveness and efficiency of code review, usually centered on the number of defects uncovered per thousand lines of code. But the clearest way is to use the two things that matter most: time and bugs.

The easiest way to visualize this is: "How much time does it take us to find and fix one bug?" I call this "Time per Bug Fix." The formula is simply: $(\text{Total Time Spent}) / (\text{Number of Defects Found})$.

This metric is clear and tangible; you can easily understand what it means and why it's interesting. It's also easy to compare to other processes, even if you're just estimating.

For example, how long does it take to find and fix a bug found in QA? A QA person has to run through some code, discover a problem, create a trouble ticket, and write down how to reproduce the problem. Then, a developer has to go find the bug—often the evidence in QA is an error dialog that doesn't help you understand the root cause of the problem—then fix it. QA then has to verify the fix before it can be considered finished.

A typical value for Time per Bug Fix for a lightweight peer review process is 5 to 15 minutes. The QA cycle described above is surely much longer than that. And let's not even talk about the damage a bug can do (in

Your potential. Our passion.®

Microsoft



A group of people are in a gymnasium, playing a game. A large green tentacle monster is in the center, holding a man in a blue shirt. A woman in a purple shirt is running towards the monster. A man in a red shirt is in the background. A yellow laser line is drawn on the floor. A colorful geometric shape is in the top left corner.

GOOD TEAMS SCORE.
FAST TEAMS SCORE OFTEN
AND DEPLOY EARLY.

The logo for Microsoft Visual Studio Team System. It features the Microsoft logo (four colored squares) above the text "Microsoft Visual Studio" and "Team System" below it.

Microsoft
Visual Studio
Team System

DEFY ALL CHALLENGES TOGETHER.

Visual Studio® Team System 2008 helps you build quality in from the opening minute, manage code from multiple sources, and keep your code base and database in line. Download a trial at microsoft.com/defyallchallenges/team

The Android Developer Experience

Is this the 'droid you've been looking for?

by Mike Riley

With the success of the Apple iPhone, a new surge of development opportunities has arisen in the consumer mobile computing space. However, due to Apple's walled garden approach, some developers have been less compelled to spend a lot of time investigating its SDK. Compounding this closed environment is Apple's insistence to exclusively base the platform's programming syntax to be the same as Apple's desktop OS language of choice, that being Objective-C. While I've written a few simple applications in Objective-C, the language never really captured my interest, and its inability to reuse the code (especially the Cocoa GUI-based elements) on non-Apple platforms further reduced my desire to learn it more deeply. While Objective-C bindings can be wrapped around scripting languages like Perl, Python, and Ruby, the amount of work necessary to wire up cross-platform unfriendly GUI's isn't worth the effort. Besides, it doesn't appear that Apple will be allowing any other programming languages to reside on its iPhone platform anytime soon.

When Google announced its Linux-based Android platform that hosted applications written using Java syntax, my mobile development interests were quickly realigned to the open platform mantra that Google was chanting. And yet, even though Google released the Android SDK earlier last year, it wasn't until the physical hardware development units became available in December 2008 that I decided to take a more serious look at this alternative mobile OS.

Signing Up

To become a privileged member of the Android Developer Program, you must first visit the Android Market and create a Gmail account if you

don't already have one. The Gmail account is required for Google Checkout, employed for the one-time \$25 charge of being granted the permission to post Android applications to the Android Market after agreeing to the lengthy developer agreement. It is also through this marketplace that developers can purchase a single G1 developer device.



Figure 1: The Android Open Handset Alliance home page provides a starting point for obtaining the Android SDK and ordering a G1 developer device.

The Hardware

Unlike the G1 model currently sold at T-Mobile and other stores, the developer G1 is an unlocked model that permits SIM cards from other carriers besides T-Mobile to be used with the device. More importantly, this G1 version offers root user access, giving you full access to the Android filesystem as well as the ability to re-flash the device without constraints. Indeed, adventurous hackers have already created a minimal Debian-based Linux distribution that can be flashed onto the G1. This delightful experiment is nowhere near ready for primetime use, but the fact that it can even be done at all will unquestionably motivate other Linux

mobile enthusiasts to charge ahead with even more innovative hacks. Needless to say, the developer unit isn't a device for the technically challenged. As a mark of special exclusivity, the developer unit G1's rubberized back plate has a custom Android polygon design imprinted across it for additional "Android Club" geek-cred points.

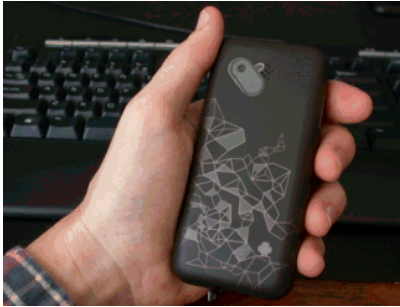


Figure 2: The back cover of the G1 dev unit sports a distinctive polygon art form advertising to other Android users of the device's elevated status.

Based on the T-Mobile G1 developed by prolific Windows Mobile hardware designer HTC and modeled upon the HTC Touch, the developer version is everything initial reviews of the consumer G1 said it was: a good first attempt with adequate single touch screen resolution, feature laden with 2G/3G, Bluetooth, GPS and WiFi radios, accelerometer, digital compass and...an abominable battery life. Since I have been using the G1, I have yet to make it through the day without charging it at least twice. However, barring the fact that this ambitious platform performs miserably in the power consumption category, it nevertheless provides a stable development reference platform to create and test Android applications in the real world.

There were other hardware issues that bothered me enough to have to correct with additional accessory purchases. First and foremost, the 1-GB microSD card included with the phone is simply too anemic by today's multimedia standards. While I debated upgrading to a 16-GB card, I opted for a much less expensive 8-GB capacity instead. Another truly bone-headed design limitation (or greedy aftermarket accessory approach, depending on the conspiracy factor) is the G1's proprietary HTC miniUSB-like adapter

that is used for both the unattractive, bulky earbuds as well as the power recharge connection. Without additional adapters, G1 users cannot, for example, listen to music and charge their device at the same time. Fortunately, an inexpensive splitter from Amazon (search for "2-in-1 mini USB adapter") solved this annoying dilemma at the expense of additional bulk clipped to the base of the phone. Finally, what seems to be a standard omission for most mobile smartphone devices these days is a protective carrying case. I decided I had invested enough in phone extras and instead simply left the

Mylar cover on that protected the screen during shipping.

Incidentally, in keeping with the burgeoning Android OS hacks, a few enterprising Linux netbook hobbyists have succeeded in compiling the Android source targeting an Asus eeePC subnotebook. While this certainly generated a buzz in the Android community, using such a device as a G1 replacement isn't practical due to the lack of hardware support such as the sensors and various drivers. Regardless, the feat further strengthens the reality of Android's open-source platform promise.

PC-lint 9.0 for C/C++

Presents
Bug of the
Month #587

Much to the chagrin of European bankers, Abraham Lincoln issued greenbacks to finance the Civil War. This program represents an early effort to detect counterfeit greenbacks. But something's going wrong. Can you spot it? Visit our web site at www.gimpel.com

```
#include <stdlib.h>
// check_serial returns true if the
// serial number is OK.
bool check_serial( unsigned serial )
{
    if( (serial & 0x2E537B4) == 0x2C42540 )
        return true;
    else return false;
}

// main() returns 0 if the serial number
// is OK and non-zero otherwise.
int main( int argc, char* argv[] )
{
    unsigned serial_no;
    if( argc <= 1 ) return 2;
    serial_no = (unsigned) atoi( argv[1] );
    if( check_serial( serial_no ) ) return 0;
    else return 1;
}
```

PC-lint for C/C++ will catch this and many other bugs. It will analyze a mixed suite of C and C++ modules to uncover bugs, glitches, quirks and inconsistencies.

Introducing Version 9.0: What, you ask, can be added to our ground-breaking lint products – products that already contain inter-function value tracking, auto variable and member value tracking, customizable function semantics, a comprehensive system of message suppression, and a rich set of over 500 Warnings that all together provide an environment in which few bugs can survive and none can flourish?

The answer to that question is found in

Version 9.0, which provides support for pre-compiled headers, static variable tracking, multi-thread analysis, support for MISRA C 2004 and MISRA C++ 2008, a rich set of program information reports including stack usage, and over 146 new messages.

**Full Support for ANSI/ISO C and C++
PC-lint for C/C++ \$389**
Numerous compilers/libraries supported.
Runs on Windows, MS-DOS, and OS/2.

FlexeLint for C/C++ Call
The same great product for all UNIX systems
Distributed in shrouded C source form.

30 Day Money Back Guarantee

Gimpel Software

Serving the C/C++ Community for 22 Years

CALL TODAY (610) 584-4261 or FAX (610) 584-4266

www.gimpel.com

PC-lint and FlexeLint are trademarks of Gimpel Software

The Software

The OS version shipping on the developer G1 that I received was an unlocked Nov 3 1.0 build running the Linux 2.6.25 kernel. While this first-generation version is a remarkable achievement for the Android team, there are unsurprisingly a few rough edges. I've read Android reviews stating how stable it is and how it's nearly crash-proof. That didn't turn out to be the case for me. I had the OS unexpectedly and forcibly reboot on me several times, most often when using Android's Music application and listening to MP3 file playback.

There is also the fact that the Android designers took the iPhone approach of masking the underlying filesystem such that files, even those being stored on the microSD card, cannot be easily accessed and manipulated from a standard file management/browser approach using the standard application set. Instead, a visit to the Android Market is necessary to more easily access the filesystem when untethered from the host development environment. The two I found most helpful are a third-party simple file browser called OI File Manager, and Google's own Terminal shell program. On consumer G1 models, the Terminal program isn't very helpful since root access is locked, but on the developer G1 model, root access is helpful, even a necessity at times. Since the version of OI File Manager I used can only affect a single file at a time, I more frequently call upon the Terminal program to perform most of my file manipulation.

More importantly, the Terminal helped

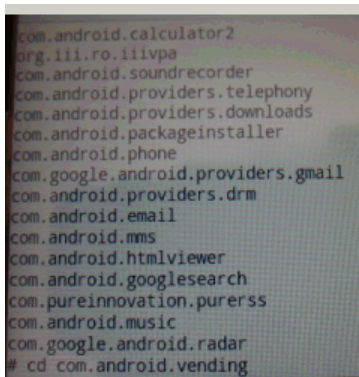


Figure 3: The Google Terminal Android application running on the G1 with root level access.

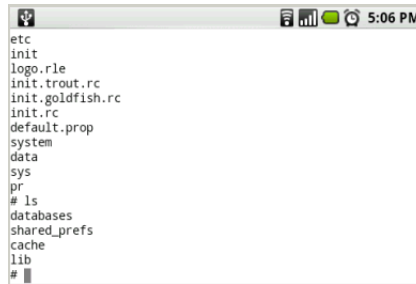


Figure 4: Screen capture of the Terminal application running on the G1, grabbed using Android's ddms developer debugging tool.

me alleviate one of the most painful design oversights of the Android Market and various Google applications like the Maps and Browser. This ravenous storage eater is the current version's inability to have these standard applications store their downloaded data caches on the microSD card and not the main onboard memory of the device. Googling for Android Market and cache together will yield a litany of angry complaint links from end users and developers alike of how their G1s are rapidly exhausted of memory as a result of these two applications lacking such an obvious feature.

As of this writing, the Android team is working on an update to the Android OS, codenamed "Cupcake," that is expected to fix this glaring oversight as well as add new features such as stereo Bluetooth (known as A2DP) support and a few other nice enhancements. In the meantime, hard-pressed developers equipped with the Terminal program have come up with a very easy UNIX-like solution to the cache problem. After downloading and installing the Terminal application from the Android Market, launch it, go to root, and create a new directory called /sdcard/marketcache (this can be named anything, and even reside in a deeper subdirectory). Navigate to the /data/data/com.android.vending directory, delete the cache folder located there (rm -R cache), and create a symlink to a new cache folder location on the microSD card (in this example, execute ln -s /sdcard/marketcache cache as root). This caching issue is a problem not just with the Android Market app, but several others as well, including the Browser, Maps, and even some third-party Android programs that prefer to keep cached

data on the device instead of providing an option to store locally cached data on removable storage. Because this is such a prevalent problem, community contributors at modmygphone.com have created a wiki entry outlining the steps needed to alleviate the cached storage constraints with various Android applications. Visit modmygphone.com/wiki/index.php/Caches_To_SD_Card for details. Note that because these cache relocations require root level access, they can only be performed easily on the developer G1 models. End consumers will either have to hold out for Cupcake, cold reset their devices to clear out the cache (and everything else they had installed), or embolden themselves with attempting to unlock the commercial G1 handset on their own and possibly breaking the phone in the process.

If you prefer not to mess with the file system and want to minimize the cache size, you do have another option. Developer Jay Freeman has created an Android Market front end at cyrket.com that anyone can visit with a web browser. This helpful site also allows non-Android owners to see what the Android Market has to offer. As long as Jay keeps cyrket.com alive, I will continue to visit it first before perusing the latest market updates via the Android Market application.

The Development Environment

Setting up the optimal configuration for Android development is a fairly easy process that takes less than 15 minutes, not including download time for the software. The Sun JDK 1.5 or higher is an obvious initial requirement to run both the Eclipse IDE and the Android Developer Toolkit (ADT) plugin set. I chose to use the latest 1.6 release. Once the JDK version requirement is met, download and install the Eclipse IDE. The nice thing about Android development is that it can be performed on Windows, Mac, or Linux, the same platforms that Eclipse runs on. Any Eclipse version 3.2 or higher will do. I chose the latest 3.4.1 Ganymede release. While the Eclipse IDE isn't absolutely essential, the Android team has written the ADT exclusively for the Eclipse environment that makes launching and managing the Android emulator much easier. There are

reports of some brave attempts to wire up these pieces within Sun's own NetBeans IDE, but unless there is some serious aversion to Eclipse, this effort appears to be more trouble than it's worth.

Next, download and install the Android SDK. This can be installed anywhere as long as the Eclipse plug-ins can consistently locate the installation path. Finally, launch Eclipse and grab the ADT via the usual Eclipse plug-in install route. Select "Software Updates" from the Eclipse "Help" menu, click on the "Available Software" tab, then the "Add Site" button and enter the location of the ADT. Restart Eclipse and enter the installation path of the Android SDK in the Window → Preferences → Android dialog box.

Windows and Mac users should be ready to go, but Linux 64-bit users will need to ensure they have the 32-bit emulation libraries (ia32-libs) installed; otherwise the proprietary, closed source emulator compiled in 32-bit code for the Linux environment will fail to run on that 64-bit host.

Check to make sure everything is correctly configured by launching Eclipse, importing the APIDemos project from the Android SDK demo folder, and running it. Doing so should launch the emulator and, once confirmed running (initial instances take between 30 seconds and a minute depending on host computing resources), select Run. This will compile the project's source code and various resources into a single Android package called an "apk" file, download and instantiate APIDemo in the emulator. Fortunately, subsequent Android runs will occur much faster as long as the emulator is allowed to keep running in the background.

The emulator itself is for the most part an accurate reflection of a real Android device like the G1. Unfortunately, the emulator used during the writing of this article could not emulate the sensors or radios such as the accelerometer or GPS. However, the aforementioned folks at OpenIntents have created a helpful alternative to help simulate such

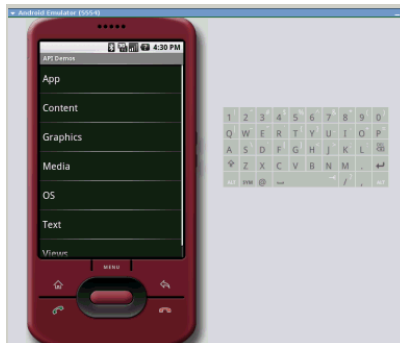


Figure 5: The Android emulator running the ApiDemos sample application.

inputs at the cost of modifying an Android project's code to get it to talk to OpenIntents's Sensor Simulator. However, I suspect that if the development community clamors loudly enough, the Android SDK team will respond with such capabilities built into future SDK releases.

Once acclimated to the emulated environment, it's time to test sending applications to the actual G1. Windows users first need to download and install the Android USB Windows driver. Linux users have more work to do. A rule has to be added to the host computer (example, for Ubuntu Hardy users, a new file needs to be created at /etc/udev/rules.d/50-android.rules containing the following: `SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"`. Once saved, `chmod a+rx` the file and the host

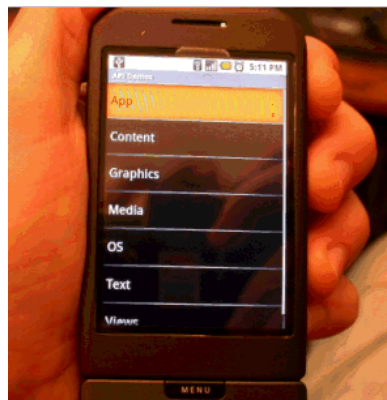


Figure 6: The G1 running the ApiDemos sample application, downloaded and instantiated from the ADK Eclipse plug-ins.

computer should be able to identify and communicate with the Android device when plugged in via the USB port. Mac users fortunately do not have to mess with any drivers or settings, as the Android SDK tools and Eclipse environment will automatically recognize the device. As Google's SDK installation documentation boasts, "it just works." And they're right — it does.

The Android plug-ins will automatically assist Eclipse in targeting the attached device and downloading and running the desired apk accordingly. Simple.

With both the emulator and device successfully receiving and running the API demo, you're ready to start writing your own Android applications.

Writing an Android Application

Creating an Android application is relatively painless, though due to the forced abstraction of resources including layout, strings for easier internationalization, and raw resources such as audio clips and images, a bit more forward thinking and manual wiring of the interface elements is required.

Assuming all the prerequisites have been met and the API demo successfully launched on either the emulator or the developer device, the Android Eclipse plug-in can be called upon to generate a skeleton Android application, complete with required class wrappers, files, and folder layouts. However, logic and interface code needs to be written and wired up before seeing anything meaningful on the Android display.

After launching Eclipse, select File → New → Project... and select Android Project. Name the project (for example, "HelloDDJ") along with a Package name (following the Java package namespace convention; for example, "com.ddj.android.hello"), Activity name (the name of the subclass from Android's *Activity* class; for example, *HelloDDJ*) and lastly, a human-readable Application Name (Hello, Dr Dobbs).

Clicking the Finish button will automatically generate the basic code and layout XML,

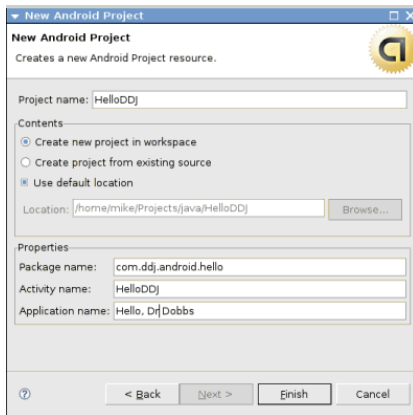


Figure 7: Creating a New Android Project with wizard assistance from the Android Developer Toolkit.

enough to execute and display a simple window with the name of the application in the titlebar and a Hello World, (followed by the Activity name, in this example's case, *HelloDDJ*) text string in the body. However, before doing so, let's first explore what files the wizard generated. Expanding the *HelloDDJ* src tree reveals two auto-generated class files:

- The *HelloDDJ.java* containing the *HelloDDJ* class, extended from Android's *Activity* class.
- *R.java* final class.

This *R* class contains pointers to string, images, layouts, audio, video and other appli-

cation assets. The *R* class is automatically managed and updated by the ADT whenever file assets are added or removed to the project's resfolder. I discovered that this *R* generator doesn't like files that begin with numbers, so name your image and audio files beginning with standard ASCII characters.

Examine the contents of the resfolder and note the three subfolders: *drawable* (containing images such as the generic program icon, as well as other bitmap graphics in PNG or JPEG formats), *layout* (containing the program's various layout definitions in XML format), and *values* (containing the program's strings structured in XML format). One folder that isn't auto-generated but required if adding file assets such as audio (AAC, AMR, MIDI, MP3, OGG, WAV, and WMA encoded files are valid Android audio playback formats) and/or video (H.263, H.264 and MP4 encoded video files are valid Android video playback formats) is the *raw* folder.

Because Android enforces strings to be referenced rather than embedded in the code, using a *System.print.out()* statement to display text in a window isn't valid. Instead, the strings are identified in the *strings.xml* file and referenced in the *Android-Manifest.xml* and various layout xml files (for example, *main.xml*). This keeps all the program strings in a single file, greatly sim-

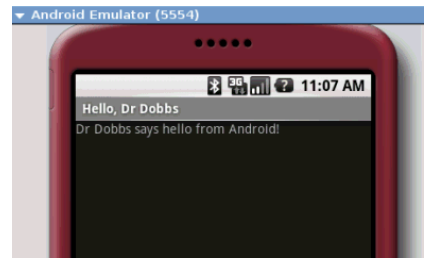


Figure 9: Dr Dobbs says 'hello' from the Android emulator.

plifying localization and generally organizing all the strings displayed in the program. Of course, the Text properties of the various Android display elements like the *TextView* class can be programmatically modified, but the static strings like those found in about boxes, forms, and dialogs are contained in the *strings.xml* file.

Thus, to modify the *HelloDDJ* string, open the *strings.xml* file, select the hello string value and replace it with "Dr Dobbs says hello from Android!" Save the project and run the application. This will fire up the emulator (or send the application straight to the G1 if it is configured and hooked up to the host computer) and display the Hello, Dr Dobbs window containing the edited welcome message.

Once the rudimentary steps of creating a simple Android application are understood, go to the next level by writing a few conceptual tests that call upon frequently accessed features, such as setting an alarm, displaying alerts, displaying a web page, or capturing images from the built-in camera. The Android SDK documentation has a section dedicated to such tips.

One of the deflating aspects of the ADT is the lack of any kind of GUI builder in the current release. Instead, each layout needs to be described in manually constructed XML. Once well formed, this XML can be rendered in the layout view to check for accuracy and positional placement. Android offers several layout models depending on the application's requirements such as screen orientation, form resizing, absolute positioning, and others. Fortunately, Android developer Brendon Burns has created both an applet and standalone Java-based design tool called "DroidDraw" that helps UI designers create

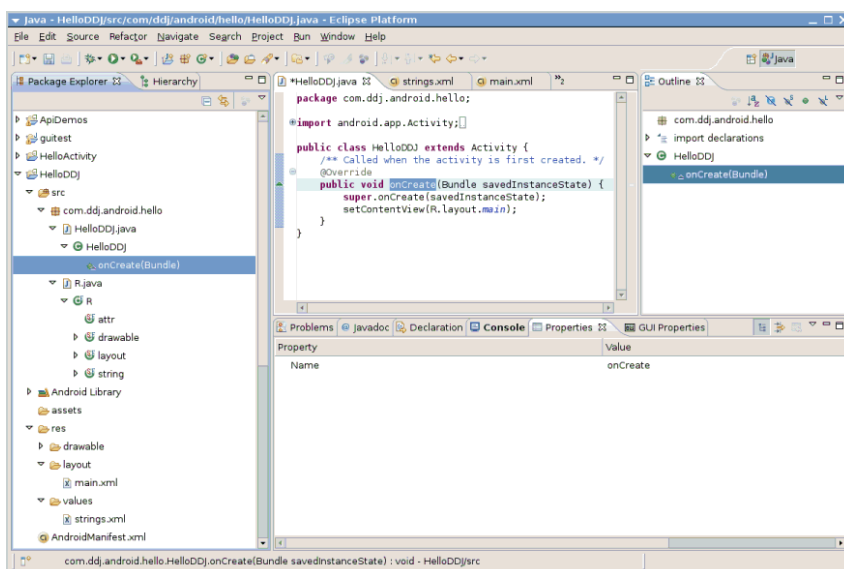


Figure 8: Examining the generated Android project files.

Android represents an attempt to satisfy both the extreme-tech and the non-tech audiences

and export valid Android XML layout files. While I expect the Android development team to eventually incorporate round-trip drag-and-drop GUI construction within Eclipse, DroidDraw supplies a more than adequate placeholder for Android UI generation.

As the Android developer community continues to expand, I expect more tools like DroidDraw to spring up, helping developers scratch the itches created by deficiencies in the SDK. And if Android's marketshare expands to the percentages projected by Google and other mobile industry analysts, there will no doubt be commercial add-ons, libraries, widgets, and code management tools created for an eager and demanding Android developer audience.

Books, Websites, Cool Apps, and Other Developer Resources

As enthusiasm for Android development continues to expand, more websites, books, and articles will add to the already adequate amount of information available today. Some of the more useful resources I discovered while working on my own Android development efforts include the following:

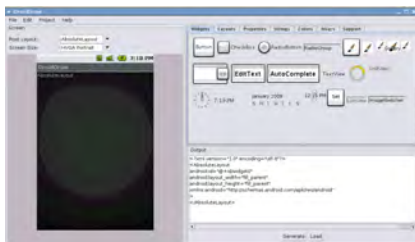


Figure 10: DroidDraw is a very useful donation-ware utility for quickly assembling Android UI layouts..

- *Android Essentials* by Chris Haseman, published by Apress. This was one of the first Android developer titles on the market, and while the content is still relevant, more recent titles have eclipsed its emulator-centric tutorials.
- *Hello Android* by Ed Burnette, published by Pragmatic Bookshelf. Just over 200 pages, this short book gets impatient developers like me up to speed quickly on programmatically controlling Android's most compelling features. Visit dobbscodetalk.com for my more comprehensive review of this title.
- *Professional Android Application Development* by Reto Meier, published by Wrox. One of the most comprehensive books on Android development available today.
- Android SDK Documentation. A first stop for anyone interested in writing Android applications. Do not pass go, do not start writing a line of code or post questions in the various Android website forums until this reference material has been read and digested.
- Android Development Community. A community-driven Android developer website chock full of reviews, tutorials, coding problem threads and other discussion groups, each of which can be subscribed to via RSS.
- Android Developers Google Groups. Google monitored developer discussion forum with over 14,000 members at the time of this article's publication.
- Google Maps (standard application on the G1). This application raised the roof at early Android demonstrations. While the compass mode in Streetview isn't the most efficient way to peruse street-level imagery, it undoubtedly continues to be the coolest.
- ShopSavvy by Big in Japan. One of the big winners in the Android Developer Challenge (code.google.com/android/adc.html), ShopSavvy uses the G1's built-in camera to capture a product

barcode, identify the product, and present best price options online and local retail outlets based on the user's GPS coordinates.

- Twidroid by Ralph Zimmermann and Thomas Marban. Twitter client with built-in geotagged photo blogging support. This mobile microblogging client is also one of the first to support Laconi.ca's PHP-based open microblogging platform.

Conclusion

I have been developing applications on various mobile platforms since the days of the original Palm OS, spending most of that timespan within various incarnations of the Windows CE environment. When the first-generation iPhone shipped without an open SDK, my desire to do anything interesting on that device quickly waned. Besides, I already had everything I needed on my Windows Mobile gadgets anyway. Of course, the iPhone was a game-changing device that brought to the less technical crowd what I had already been doing for years.

Android represents an attempt to satisfy both the extreme-tech and the non-tech audiences. I suspect that for the time being, Android will serve code-hungry developers for a while but will struggle to gain significant marketshare in the segment that Apple's iPhone is serving so effectively. But as a result of the open conversations and open platform, faster innovation is more likely on Android compared to any other modern day mobile OS. How this will translate into commercial ventures is still murky, but from a pure development standpoint, I can't recall when I had this much fun (coupled with a few brief bouts of frustration) writing applications for a mobile device. Even with its first-generation flaws, the Android is the smartphone that currently goes in my pocket every morning, leaving my other mobile devices behind to collect dust on the retired gadget shelf back home.

—Mike Riley is a Dr. Dobb's Contributing Editor.

Your best source for software development tools!

programmer's paradise®



LEADTOOLS Document Imaging v16:

by LEAD Technologies

LEADTOOLS Document Imaging has every component you need to develop powerful image-enabled business applications including specialized bi-tonal image display and processing, document clean up, high-speed scanning, advanced compression (CCITT G3/G4, JBIG2, MRC, ABC) and more.

- Multi-threaded OCR/ICR/OMR/MICR/Barcodes (1D/2D)
- Forms recognition/processing
- PDF and PDF/A
- Annotation (Image Mark-up)
- C/C++, .NET, WPF - Win32/64

Paradise #
L05 03201A01
\$2,007.99

programmers.com/lead

dtSearch Engine for Win & .NET

Add dtSearch's "blazing speeds" (CRN Test Center) searching and file format support

- dozens of full-text and fielded data search options
- file parsers/converters for **hit-highlighted** display of all popular file types
- Spider supports dynamic and static web data; **highlights hits** with links, images, etc. intact
- API supports .NET, C++, Java, SQL and more; new .NET Spider API

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second."
—InfoWorld

New
64-bit
Version!



Single Server
Paradise #
D29 02101A07
\$949.99

programmers.com/dtsearch

VMware Infrastructure Acceleration Kits

New
Release!

VMware Infrastructure 3 offers SMB organizations a scalable and cost-effective way to optimize utilization of technology assets, simplify IT management and protect the data and IT environments that run their businesses. Three new Acceleration Kits are now available, one for each of the three editions of VI3. These new kits offer midsize and smaller organizations and branch offices a cost-effective way to deploy a comprehensive virtualization solution that includes centralized management functionality. The ideal organization has a growing IT environment with between 15-60 servers.

Call for pricing on the Standard High Availability Acceleration Kit and Midsize Acceleration Kit.



Foundation
Acceleration Kit
Paradise #
V55 47101A01
\$2,583.99

programmers.com/vmware



/n software Red Carpet Subscriptions

by /n software

/n software Red Carpet™ Subscriptions give you everything in one package: communications components for every major Internet protocol, SSL and SSH security, S/MIME encryption, Digital Certificates, Credit Card Processing, ZIP compression, Instant Messaging, and even e-business (EDI) transactions. .NET, Java, COM, C++, Delphi, everything is included, together with per developer licensing, free quarterly update CDs and free upgrades during the subscription term.

Paradise #
D77 09101101
\$1,319.99

programmers.com/nsoftware

c-treeACE™ Professional

by FairCom

The c-treeACE database engine is a high performance database alternative proven by developers in mission critical enterprise systems, desktop deployments, and embedded devices for over 25 years.

- Complete set of APIs including ADO.NET, LINQ, C#, C/C++, ODBC, JDBC, VCL, and dbExpress
- Graphical productivity tools
- Simple deployment
- No DBA or ongoing administration
- Low deployment licensing costs
- Cross-platform support for all major platforms including Windows, UNIX, Linux, and Mac OS X

Make your applications faster, easier to deploy, and more affordable with c-treeACE.



Paradise #
F01 0131
\$711.99

programmers.com/faircom

TX Text Control 14

Word Processing Components

TX Text Control is royalty-free, robust and powerful word processing software in reusable component form.

- .NET WinForms control for VB.NET and C#
- ActiveX for VB6, Delphi, VBScript/HTML, ASP
- File formats DOCX, DOC, RTF, HTML, XML, TXT
- PDF export without additional 3rd party tools or printer drivers
- Nested tables, headers & footers, text frames, bullets, numbered lists, multiple undo/redo, sections, merge fields
- Ready-to-use toolbars and dialog boxes

New
Release!



Professional Edition
Paradise #
T79 02101A01
\$919.99

Download a demo today.

programmers.com/themagingsource



DevTrack Small Team Edition

Powerful Defect and Project Tracking

by TechExcel

TechExcel DevTrack is the most powerful, affordable and easy-to-use defect and project tracking tool for development organizations. You'll dramatically transform your development processes, save significant time and resources, and deliver quality products on-time and on-budget.

- Sophisticated workflow engine
- Point-and-click administration
- Fully configurable user interface

5-User Pack
Paradise #
T34 0208
\$1,414.99

programmers.com/techexcel

Altova® MapForce® 2008

Visual Data Conversion, Transformation, and Integration Tool

by Altova

MapForce: The premier data mapping, conversion, and integration tool from the creators of XSLSpy®. Through its visual interface, users can map seamlessly between any combination of XML, database, flat file, EDI, and/or Web service, then convert data instantly or auto-generate an application for recurrent transformations. Languages for code generation include: XSLT 1.0/2.0, XQuery, Java, C++, and C#.



ALTOVA®
mapforce®

New
Release!

Enterprise Edition
1 user
Paradise #
IOD 03101A02
\$1,169.99

programmers.com/altova

Enterprise Architect 7.1

Visualize, Document and Control Your Software Project by Sparx Systems

Enterprise Architect is a comprehensive, integrated UML 2.1 modeling suite providing key benefits at each stage of system development. Enterprise Architect 7.1 supports UML, SysML, BPMN and other open standards to analyze, design, test and construct reliable, well understood systems. Additional plug-ins are also available for Zachman Framework, MODAF, DoDAF and TOGAF, and to integrate with Eclipse and Visual Studio 2005/2008.

New
Release!



Corporate Edition
1-4 Users
Paradise #
SP6 0001
\$182.99

programmers.com/sparxsystems



StorageCraft ShadowProtect IT Edition v3.x

by StorageCraft

Create, edit or restore backup images on as many servers, desktops and laptops as needed. Create online or cold state backups in minutes, no software installation required. StorageCraft® ShadowProtect IT Edition provides complete bare metal recovery in minutes. ShadowProtect IT Edition provides IT Professionals with a bootable Windows environment to create and restore compressed and encrypted backups, no software installation required.

Paradise #
SC5 03101A01
\$3,252.99

programmers.com/storagecraft

Telerik RadControls

by Telerik

Add grid, combo, editing, navigation and charting functionality to your AJAX and ASP.NET projects. RadControls for ASP.NET enhances your Web applications by adding AJAX functionality to your ASP.NET projects. The suite takes full advantage of the features included in Visual Studio 2005. RadControls for ASP.NET helps developers deliver feature-rich, standards-compliant (WAI-A, WCAG 1.0, XHTML 1.1) and cross-browser compatible Web applications, while significantly cutting their development time. RadControls for ASP.NET includes: RadEditor, RadTabstrip, RadInput, RadCalendar, RadUpload, RadWindow, RadAjax, RadGrid, RadCombobox, RadMenu, RadSpell, RadChart, RadTreeView and more.



Single Developer
Paradise #
TB3 01101A01
\$648.99

programmers.com/telerik

Multi-Edit 2008

by Multi Edit Software

Multi-Edit 2008 delivers, a powerful IDE, with its speed, depth, and support for over 50 languages. Enhanced search functions include Perl 5 Regular Expressions and definable filters. Supports large DOS/Windows, UNIX, binary and Mac files. File Sync Integration for: Delphi 6, 7, 2005, C++Builder 6, BDS 2006 and RadStudio 2007, VB 6, VC 6, VS 2003 & VS 2005. Includes file compare, code beautifying, command maps, and much more.

New
Release!



1-49 User
Paradise #
A30 01201A02
\$161.99

programmers.com/multiedit

800-445-7899

Prices subject to change. Not responsible for typographical errors.

programmersparadise.com

Internet Explorer 8 and Compatibility Views

Beta 2 of Internet Explorer 8 passes the Acid2 test

by Dino Esposito

Acid2 is a sample page available at www.webstandards.org/action/acid2 created with the precise goal of helping browser vendors to test their product against proper support for latest web standards. Few of the currently used web browsers have passed the test. Among the others, I'd like to mention Safari, iCab, Konqueror, Opera, and Firefox 3. What about Internet Explorer?

Internet Explorer 8—only in beta 2 at the time of this writing—has passed the Acid2 test; and it is the first version of the Microsoft's browser to make it. The IE8 team worked out a completely new rendering engine that now follows the CSS 2.1 specification as closely as possible.

This new rendering mode is enabled by default for pages hosted in Internet web sites with the clear purpose of giving users the best possible viewing experience. Unfortunately, such a great viewing experience is only possible if web pages are created according to the latest web standards. This is not often the case. To avoid breaking so many Web pages, a Compatibility View mode has been added to Internet Explorer 8.

The Compatibility View feature works in a rather intuitive manner. It simply switches back the rendering engine of Internet Explorer 8 to version 7. In this way, from a rendering perspective you are just not using a newer browser. As a result, all sites not explicitly redesigned and restyled for latest standards will display correctly.

How would you activate the Compatibility View mode? As mentioned, the Compatibility mode is disabled by default on Internet web pages, but it is the default for intranet sites. Large organizations can keep on using their applications as usual even when they upgrade to Internet Explorer 8. In this way, they can plan site updates ahead and proceed with that at their earliest convenience.

Changing the settings of the Compatibility View is possible interactively, too. A new UI button is displayed in the navigation bar, at the right end of the address bar and near to the Refresh button. It is worth noting that the button is hidden when you are in the Compatibility View mode and only

appears when the page is being viewed through the new Standards mode engine.

As a web developer, you can force a given page to use the new rendering mode or stick to the Internet Explorer 7 rendering mode. All you need is a simple `<meta>` declaration like the one shown below:

```
<meta http-equiv="X-UA-Compatible" content="IE=8" />
```

You place the tag into the `<head>` section of the page. As a result, Internet Explorer 8 will render the page using the new Standards mode. Likewise, you can programmatically force a legacy rendering through a slightly different `<meta>` tag:

```
<meta http-equiv="X-UA-Compatible" content="IE=Emulate7" />
```

The X-UA-compatible header is not case sensitive. It poses other constraints though. In particular, it must appear in the `<head>` before all other elements, except for the title element and other `<meta>` tags.

If you don't want to edit each and every web page, you can resort to adding a custom header to the site configuration. For IIS 7, the following web.config script suffices:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <httpProtocol>
      <customHeaders>
        <clear />
        <add name="X-UA-Compatible" value="IE=EmulateIE7" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

For IIS 6, instead, you resort to the user interface of the IIS Manager. Finally, you can check the rendering engine via JavaScript. You use the `documentMode` property on the document object.

```
function IsStandardsMode()
{
  !if (document.documentMode)
    return false;
  return (document.documentMode == 8);
}
```

The property exists only in Internet Explorer 8 and is undefined for other browsers. It takes a value of 8 if the newest rendering engine is active.

—Dino Esposito is a trainer and software consultant based in Rome, Italy.

The Art of Application Performance Testing

A wealth of hard-earned lessons

Reviewed by Mike Riley

The Art of Application Performance Testing

Ian Molyneaux

O'Reilly Media

158 pp., \$34.99

ISBN: 978-0-596-52066-3

Don't be fooled by the deceptively small page count of this book. Author Ian Molyneaux has distilled a wealth of hard-earned lessons into a set of checklists and bullet points that crystalize the majority of application performance testing principles in real-world practice today. Read on for a full review.

Coming in at just over 150 pages, *The Art of Performance Testing* is nevertheless a valuable read for any developer and/or tester bestowed with the responsibility of ensuring an optimal application execution experience. As is always the case in theoretically successful application construction and testing, performance expectations and realistic limits should be clearly defined before the first line of code is written. What distinguishes Ian's book apart from other, more verbose books on the subject is his advocacy for the performance testing business case and, in the book's third chapter, two case studies that show the lifecycle of such real-world testing outcomes.

The book opens with a chapter explaining the historical rationale behind performance testing, including the establishment of performance measures and outcomes, and why leaving performance testing to the last leg of an application development cycle is a really bad idea. This is followed by the largest chapter in the book and covers the fundamentals of application performance testing, the selection of performance measurement tools, constructing the test environment and setting realistic performance targets via business process owner consensus, key performance targets (such as availability/uptime, concurrency/throughput and response time), making sure an application is stable enough to test (obtaining a frozen code build, specifying the transaction list, and replay validation), and ensuring high-quality, accurate, and valid performance design and data capture.

Chapter 3 reviews best practice model approaches to performance testing via Proof of Concept (POC) checklists and form requirements via a six-step process. This chapter also reviews two real-world case studies, one for an online banking analysis and the other for a

call center. Both of these are invaluable to see how the principles and techniques promoted by the author are put into practice.

The next chapter examines the recommended statistical and analytical procedures a tester should consider when interpreting the real-time and post-captured results. Linear, knee-spikes, standard distribution curves are some of the more frequent data views expected when monitoring Key Performance Indicators (KPIs). The chapter closes with helpful pre through post task checklists sure to make performance evaluations more meaningful and relevant.

The final chapter in the book presents a rapid overview of some of today's newest application frameworks and recommended ways for dealing with measuring their performance. Rich Internet Application (RIA) technologies like AJAX, Windows Presentation Foundation (WPF), Adobe's Flash/Flex and enterprise systems like SAP, Oracle, and SOA are very briefly touched upon. The book closes with 5 appendixes, spanning from transaction examples to KPI templates showing three screen grabs of performance trees, one which erroneously displays a duplicate of the Citrix MetaFrame tree instead of the intended Microsoft IIS tree. As of this review post, no replacement image has yet been posted on the O'Reilly errata web page for the book.

In summary, while *The Art of Application Performance Testing* is rather expensive for its short notebook on status, there are a couple of gold nuggets of hard-earned experience that can be uncovered in the book and practiced in your own application development lifecycles immediately. More cost-conscious readers may find plenty of free material with nearly as good checklists and recommendations by Googling the various performance testing blogs and even several articles on the *Dr. Dobb's* main website, but for those looking for a less time-consuming and more accurate, trustworthy source of practice and experience, this book delivers the goods in considerably less time and with greater impact.

—Mike Riley is a *Dr. Dobb's* Contributing Editor.

Q&A: Parallel Programming

When it comes to parallel computing, performance can be a balancing act

by Jonathan Erickson

Parallelism and performance go hand-in-hand. But achieving maximum performance can be a balancing act, as Intel senior engineer James Reinders explains to Dr. Dobb's editor-in-chief Jonathan Erickson.

Dr. Dobb's: Sandia National Labs engineers say that adding cores to high-performance computers slows data-intensive apps. Your thoughts?

Reinders: It is an issue of "balance." It certainly isn't that data-intensive application cannot use parallelism. Good performance requires a characteristic we call "balance." One person's balance is another person's headache. Supercomputer systems, in order to get the "balance" they need, place a high premium on things like memory bandwidth, I/O bandwidth, and low latency communications to mention a few. If we have a system which is tipping out of balance in a way where computing ability is not the bottleneck (say memory bandwidth is the limitation)—then adding more computing ability (e.g., more cores) will not help. In fact, because it is likely to add more contention it can slow things.

The latest Intel architecture, we code named "Nehalem" and now available as processors known as Intel Core i7, put enormous emphasis on adding features to maintain or add balance. If the new architecture had simply added cores, without the new features, many people would have found slow down issues. The key features which add "balance" are QuickPath Interconnect (QPI) and an integrated memory controller, resulting in unprecedented amount of aggregate bandwidth.

Is it enough to get balance for everyone? It never is. Balance is a fundamental challenge in computer design, a classical example of a design goal in the face of numerous engineering tradeoffs. It will never end. That's why characterization of behavior, like the work done at Sandia, help frame the problems so they can be solved.

Dr. Dobb's: Microsoft's Azure doesn't support multi-core applications. Is there a place in the cloud for parallelization?

Reinders: Parallelism is very important to cloud computing. Azure is clearly designed to use lots of cores. The initial implementation of Azure chose to assume we will have lots and lots of cores over time, and it has a simple one-to-one correspondence of physical cores to VMs. This is because Azure's future is more about doing many things in parallel rather than doing a few things each being parallelized. In this model, a multi-threaded program can still run on a single physical core—and can still get parallelism when that core has hardware threading (such as in Intel Core i7).

My best guess would be that as we get many more cores, Azure would be tempted to let a single VM own multiple cores. I do think they will resist letting a single core run multiple VMs. To my understanding, that is the main they started with the one to one correspondence of VMs and cores. I think that was a rather bold but important decision. It add predictability to a system, and relies on growing upon a future where cores will be even more numerous.

Cloud computing loves cores. On multiple levels, it is trying to avoid shared mutable state, within applications. This is a good thing.

Dr. Dobb's: Shared-memory programming or message passing? Which is the better way to do parallel programming?

Reinders: If I have to pick—the answer is message passing. If I was teaching parallel

programming, I would start with something based on message passing.

Dr. Dobb's: Transactional memory. What is it and why is it important?

Reinders: Transactional memory tries to make “shared mutable state” in parallel pro-

Start with “Think Parallel.” Understand where parallelism can be exploited in your application, and the pros and cons of different approaches

gramming to new minds, with the goal of awakening them to “think parallel”, I would start with something based on message passing.

However, if I'm giving advice to someone with an existing program that needs parallelism added with minimal fuss and on a schedule—I'm unlikely to advocate message passing.

Message passing is a fundamental concept that is probably more important than OOP to the future of computing. I'm not sure we'll call it message passing for long.

That said, shared-memory programming isn't going away and will dominate programming for the foreseeable future. It is familiar and incremental. It is also the root of the key headache in parallel programming: “shared mutable state.” Most programmers will use shared-memory programming, and need to learn a lot about “shared mutable state” and how to deal with it. Issues that it leads to such as deadlocks and race conditions haunt parallel programmers using shared memory programming.

Message passing, on the other hand, avoids “shared mutable state”, but lacks the familiarity of shared memory programming. New frameworks, similar to OOP in many ways, which use “actors” or “agents” seem to have an intuitive feel that may attract more programmers than the most used messaging

passing interface called MPI. The concept of a “transaction” for a database is well established—and taken for granted that you either want to update a whole record in a database all-at-once or not-at-all. Imagine changing your plane ticket from Tuesday 8:00 to Thursday 11:30 but the database update switched Tuesday -> Thursday and 8:00 -> 11:30 independently, with the possibility one succeeds and the other fails. No one expects that with a database.

So transactional memory is this idea “Hey! What if we could update memory like a database—in transactions?” This would bring easier to understand programming as well as less contention between parallel threads which will lead to more scaling/speed-up. Perfect in a parallel world. This is very attractive because “shared mutable state” is the root of many parallel programming problems. That said, we just have to figure out how to build it. Key topics include: what should be in software vs. hardware?, how large of a transaction needs to be handled?, what are the exact semantics?, what are the failure modes?

It remains an incredibly important research topic because it seems to offer so much hope. There are a number of companies doing software and hardware imple-



James Reinders

mentations to continue exploring this topic. Our Software Transactional Memory (STM) Compiler is available at <http://whatif.intel.com> for experimenting in C/C++.

Dr. Dobb's: We keep hearing that multicore and parallelization will demand new skill sets on the part of developers. What kind of skills, for instance?

Reinders: I have to start with “Think Parallel.” What did a developer for programs before parallelism needed to know? The fundamentals include data structures, algorithms, modularity, parsing, databases, and operating systems. How do I mention good design and change control? How much you know about each, and what exact skills you develop, is an individual thing.

Add parallelism, and we need to understand parallelism as part of all of these. Some of this has been going on for some time—any database expert is going to have thought about parallelism in many ways for years now. Ditto for operating systems, at least at some levels. Other things like parallelism in algorithms and data structures are new to many developers.

Start with “Think Parallel.” Understand where parallelism can be exploited in your application, and the pros and cons of different approaches.

In addition to “Think Parallel” the two key skills are:

- Writing scalable code. The shift in mindset here is fundamental: a good program needs to scale much more than it needs to be efficient. This is a very difficult concept for most to embrace, and if you develop a skill at seeing how to write code that scales, you will have a

skill that will be in demand. To survive, we all need to cope with the issue of scaling some.

- Dealing with shared information, managing global state sharing. Do you share mutable state? if not, how do you update state globally? The desire to make global state visible, but not globally changeable, is really similar to the debates in OOP about public vs. private with all the same issues.

Dr. Dobb's: Locks. Good, bad, or just ugly? Putting this another way, do you prefer locks or no-locks?

Reinders: Neither, I prefer working programs that scale.

Unfortunately, as ugly as locks are, they are here to stay. Nothing will make them disappear—every if STM or lock-free algorithms catch on, locks will remain. So we need to understand them, even while we try to minimize their use. The key being “minimize”—like the Albert Einstein’s quote, “Make everything as simple as possible, but not simpler.”

Locks are not the real issue. The issue is highly contended locks. Grabbing a lock, and releasing it, generally is very clear programming to keep share mutable state correct in a parallel program. A huge problem occurs when many parts of a program have to wait for a lock to be available because there is contention. This kills scaling of a program (because there is a lot of overhead to being parallel—overhead being the waiting for locks).

Of course, the solution is eliminating the contention! As usual, the best approach is to question the need for the contention in the first place. Can you rewrite your program to not have to fight over this shared state?

If you need the contention over the shared mutable state, then you question the need for the lock. Two promising approaches are lock-free algorithms and transactional memory. The first is hard and easy to get wrong, and the later is still a research topic.

I love looking at lock-free algorithms/methods. Honestly, most lock-free proposals I see are wrong. The ones that

are right, are brilliant but can really make your head hurt as you puzzle out if they are guaranteed to work in all cases.

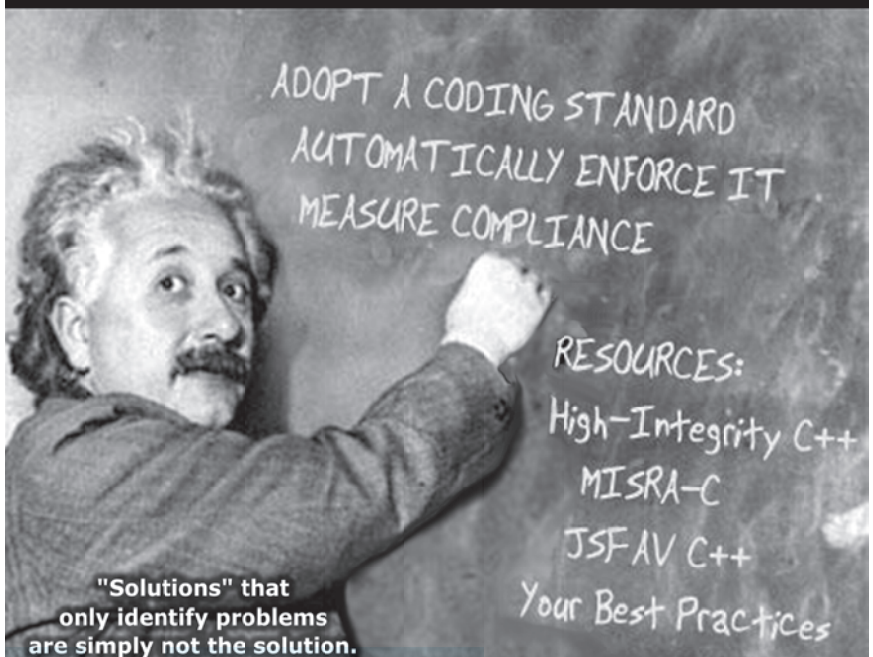
Writing lock-free code is playing with fire. The rewards for getting it right are great, the challenge many. I’ve never encourage people to try to solve there lock problems by rewriting as a lock-free algorithm. Over time, we’ll add lock-free algo-

rithms and algorithms with low contention for locks, into our libraries and programming languages. So if you want to write lock-free code, do it for all of us, and then I can point to it as something everyone should use.

So, when looking at an algorithm, don’t ask “are they lock free?” Instead ask “how much contention will I get using these?”

Intellectuals solve problems. Geniuses prevent them.

— Albert Einstein



THINK PREVENTION GET QUALITY

PRQA
Programming Research
THE CODING STANDARD EXPERTS

WWW • PROGRAMMINGRESEARCH • COM

Use Threads Correctly = Isolation + Asynchronous Messages

“Up-level” this low-level tool

by Herb Sutter

Explicit threads are undisciplined. They need some structure to keep them in line. In this column, we’re going to see what that structure is, as we motivate and illustrate best practices for using threads—techniques that will make our concurrent code easier to write correctly and to reason about with confidence.

Where Threads Fit (and Thread Pools, Sometimes)

In [1], I described the three pillars of concurrency. The first two pillars summarize the two main kinds of concurrency we need to be able to express: 1.

Keep things separate, so that independent parts of the program can run asynchronously. 2. Use more cores to get the answer faster using data-parallel and similar techniques. (The third pillar is about controlling concurrency once it has been expressed, using tools like locks and atomics.)

Table 1 summarizes these two pillars, and also summarizes how well each is served by four major tools at our disposal today for expressing concurrency: threads, thread pools, work stealing run-times, and data-parallel facilities like OpenMP.

Threads are about expressing Pillar 1 only, and this article will focus on that column: How to effectively use today’s tools, notably threads and in

	I. Isolation Via Asynchronous Agents	II. Scalability Via Concurrent Collections
Tagline	Keep things separate	Re-enable the free lunch
Summary	Stay responsive and avoid blocking by running tasks in isolation and communicating via messages	Use more cores to get the answer faster by running operations on groups of things; exploit parallelism in data/algorithm structures
Examples	GUIs, web services, background print/compile, pipelining	Trees, quicksort, compilation
Threads	OK (primary, Options 1 and 2): For independent work that is long-running (e.g., background worker) or likely to wait/block	No: If you’re using threads for this, chances are you’re trying to write your own thread pool
Thread pools	OK (secondary, Option 3): A short-running thread that rarely waits/blocks can instead “rent-a-thread” by running as a pool work item	OK: Medium overhead (context switch per work item), medium flexibility
OpenMP	No: OpenMP tasks are not guaranteed to run asynchronously (on a different thread)	OK: Low overhead, low flexibility (intended principally for parallelizing Fortran and C loops)
Work stealing pools (Cilk, TBB, PPL, TPL, PLINQ, Fork/Join)	No: Tasks are not guaranteed to run asynchronously (on a different thread)	Best: Low overhead, high flexibility

Table 1: Expressing two kinds of concurrency.

some cases thread pools, to express independent work. (We'll look at Pillar 2 in a future article.)

Threads: In a Nutshell

Here are the key things to know about threads:

- Threads are for expressing asynchronous work. The point of being asynchronous is to let the units of independent work in the application all run at their own speeds and better tolerate each other's latency.
- Threads are a low-level tool. Threads are just "sequential processes that share memory," and that kind of freewheeling anything-goes model doesn't provide any abstraction or guard rails to make good practices easy and bad practices hard. As aptly criticized by Edward Lee in his paper "The Problem with Threads" [2], threads let you do anything, and do it nondeterministically by default.
- "Up-level" them by replacing shared data with asynchronous messages. As much as possible, prefer to keep each thread's data isolated (unshared), and let threads instead communicate via asynchronous messages that pass copies of data. This best practice inherently encourages writing threads that are event-driven message processing loops, which gives inherent structure and synchronization and also improves determinism:

```
// An idealized thread mainline
//
do {
    message = queue.Receive();
    // this could block (wait)
    // ...
    // handle the message
    // ...
} while( !done );
// check for exit
```

Ideally, each thread's logic should be built around the model of servicing its message queue, whether a simple FIFO queue or a priority queue (the latter if some messages should be given priority even if they arrive later).

- Highly responsive threads should not perform significant work directly. Some threads are responsible for interacting with the user (e.g., GUI threads) or with other processes or machines (e.g., socket and communications threads), or for other reasons need to respond to messages quickly. Such threads should per-

form nearly all their work asynchronously by posting the work to one or more helper threads or, where appropriate, to a pool thread. In particular, highly responsive threads should never block by waiting for a message or trying to acquire a lock.

A final note: In some cases, a thread that is short-running and will not block (wait idly for other events, including inbound messages or locks) can be expressed instead as a thread pool work item for efficiency, to avoid the overhead of creating a new thread from scratch. That's the one valid use of a thread pool for Pillar 1.

Example 1: GUI

GUI threads are a classic example of message pump-driven code. A GUI program is driven by queued events coming in from the user or elsewhere, and its code is organized as a set of responders providing the right thing to do in response to anything from the user pressing a button (e.g., Save, Print, search and replace) to a system event (e.g., window repaint, timer pulse, file system change notification). Many GUI systems offer priority queueing that lets later messages execute sooner even if other messages in the queue arrived earlier (e.g., *SendMessage* vs. *PostMessage* on Windows).

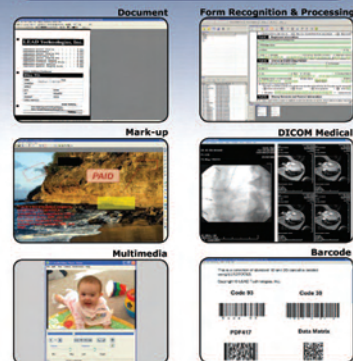
Consider the following simplified GUI pseudocode:

```
// Example 1: Sample synchronous GUI
// (not recommended)
//
while( message = queue.Receive() ) {
    // this could block
    if( it's a "save document" request ) {
        TurnSavingIconOn();
        SaveDocument();
        // bad: synchronous call
        TurnSavingIconOff();
    }
    else if( it's a "print document" request ) {
        TurnPrintingIconOn();
        PrintDocument();
        // bad: synchronous call
        TurnPrintingIconOff();
    }
    else
        ...
}
```

Running nontrivial work synchronously on the GUI thread like this is a classic mistake, because GUIs are supposed to be responsive, and to be responsive requires not just responding to events but responding to

LEADTOOLS v.16 - .NET, WPF, WCF, WF, C API, C++ Class Lib, COM & more!

Develop your application with the same robust imaging technologies used by **Microsoft, HP, Sony, Canon, Kodak, GE, Siemens, the US Air Force and Veterans Affairs Hospitals**. LEADTOOLS provides developers easy access to decades of expertise in color, grayscale, document, medical, vector and multimedia imaging development. Install LEADTOOLS to eliminate months of research and programming time while maintaining high levels of quality, performance and functionality.



- **Image Formats:** All industry standards including TIFF, EXIF, PDF, JPEG/JPEG2000, DICOM, PCL, DWG and hundreds more.
- **Image Compression:** From standard JBIG, JBIG2, ABIC, JPEG and CCITT G3/G4 to LEAD's specialized ABC, CMP and CMW compression.
- **Display Controls:** Scroll, zoom, pan, magnify glass, brightness/contrast/gamma, window level.
- **Image Processing:** 200+ filters, transforms and color conversion functions with extended grayscale (10 to 16-bit) support.
- **OCR/ICR/OMR:** Full page or zonal multi-threaded recognition with formatted output including PDF, DOC and TXT for Win32 and x64.
- **Barcode:** Read/write all industry standard 1D and 2D barcodes (DataMatrix, PDF417, MicroPDF417, QR Code and more).
- **Forms Recognition and Processing:** Automatically identify forms and extract user filled data.
- **Document Cleanup/Preprocessing:** Deskew, despeckle, line and border removal, registration marks and more.
- **PDF and PDF/A:** Read/write raster and text searchable PDF files.
- **Annotations:** Interactive UI for document mark-up, redaction and image measurement (including support for DICOM annotations).
- **Grayscale Imaging:** Display and process signed/unsigned 10-16 bit, 32 bit data.
- **Medical Web Viewer Framework:** Plug-in enabled framework to quickly build high-quality, full-featured, web-based medical image delivery and viewer applications.
- **Medical Image Viewer:** High level display control with built-in tools for image mark-up, window level, measurement, zoom/pan, cine, and LUT manipulation.
- **DICOM:** Full support for all IOD classes and modalities defined in the 2008 DICOM standard (including Encapsulated PDF/CDA and Raw Data).
- **DICOM Communications:** Full support for DICOM messaging and secure communication enabling quick implementation of any DICOM SCU and SCP services.
- **WPF (XAML):** Viewer, Image List, Annotations, Transitions and WIC Codecs.
- **AJAX:** Web Form Image controls for developing rich web applications.
- **JPIP:** Client and Server components for interactive streaming of large images and associated image data using the minimum possible bandwidth.
- **Scanning:** TWAIN 2.0 and WIA (32 and 64-bit), autodetect optimum driver settings for high speed scanning.
- **DVD:** Play, create, convert and burn DVD images.
- **Multimedia:** Capture, play, stream and convert MPEG, AVI, WMV, MP4, MP3, OGG, ISO, DVD and more.

**LEAD
TECHNOLOGIES**

800 637-1840
Free 60 Day Evaluation!
www.leadtools.com/ddjbs

LEADTOOLS SDKs feature LEAD's
iCompress™ Technologies.

them quickly. What if the GUI thread is busy processing a *PrintDocument* for several seconds, and in the meantime the user tries to resize or move the window? The resize or move request will be dutifully enqueued as a message that waits to be processed in its turn once the printing is complete; but in the meantime the user sees no effect and may try the same action, or give up and try something else instead, only to have all of the potentially duplicated or contradictory commands performed in sequence when the GUI thread is finally able to service them—often with unintended effects. Synchronous execution leads to poor responsiveness and a poor user experience.

Threads that must be responsive should never execute high-latency work directly. This includes not just work that may take a lot of processing time, but also work that might have to wait for another thread, process, or computer—including communications or trying to acquire a lock.

There are three major ways we can run high-latency work asynchronously to move it off a thread that needs to stay responsive. Using the GUI thread as a case in point, let's examine them in turn and consider when each one is appropriate.

Async Option 1: Dedicated Background Thread

Our first option for moving work off the GUI thread is to execute it instead on a dedicated background worker thread. Because there is exactly one background worker for all the GUI-related grunt work, one potentially desirable effect is that the background work will be processed sequentially, one item at a time. This can be a good thing when two pieces of work would conflict (e.g., want to use the same mutable data) and so benefit from running sequentially; it can be a

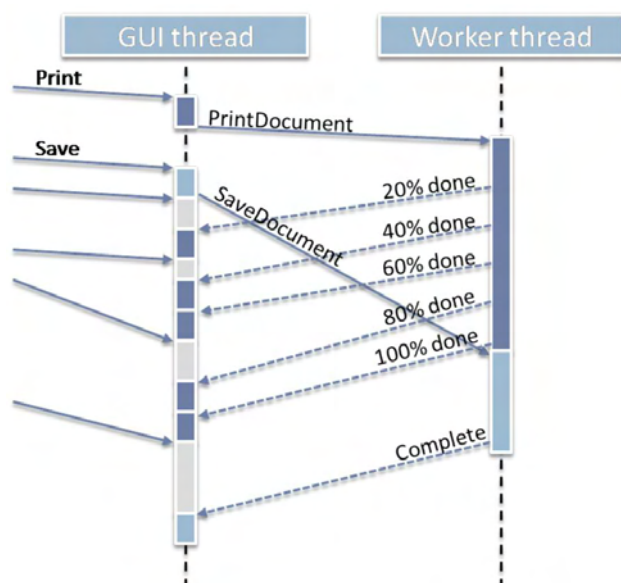


Figure 1: Getting work off the GUI thread using a dedicated background worker.

drawback when earlier tasks block later tasks that could have run sooner independently (see Option 2 below). Note also that messages don't have to go only one way: The asynchronous work can send back notifications ranging from a simple "I'm done" to intermediate information like progress status.

Figure 1 shows an example of this arrangement, where the user presses Print (dark blue) and then Save (light blue), and the two pieces of work are delegated to the worker where they can run asynchronously while the user continues to perform other operations such as moving windows or editing text (gray).

Let's consider two common ways to express the code for this option. First, we can arrange for the GUI thread to send tags representing the work that needs to be done:

```
// Option 1(a): Queueing work tags for a dedicated
// background thread. Suitable where tasks need
// to run sequentially with respect to each other (note
// corollary: an earlier task can block later tasks).
//
// GUI thread
while( message = queue.Receive() ) { // this could block
    if( it's a "save document" request ) {
        TurnSavingIconOn();
        worker.Send( new SaveMsg() ); // send async request
    }
    else if( it's a "save document" completion notification ) {
        TurnSavingIconOff(); // receive async notification
    }
    else if( it's a "print document" request ) {
        TurnPrintingStatusOn();
        worker.Send( new PrintMsg() ); // send async request
    }
    else if( it's a "print document" progress notification ) {
        if( percent < 100 ) // receive async notification
            DisplayPrintPercentComplete( percent );
        else
            TurnPrintingStatusOff();
    }
    else
        ...
}

// Dedicated worker thread: Just interprets
// the tags on its queue and performs the
// appropriate action for each tag.
//
while( message = workqueue.Receive() ) { // this could block
    if( it's a "save document" request )
        SaveDocument(); // now sends completion notification
    else if( it's a "print document" request )
        PrintDocument(); // now sends progress notifications
    else
        ... // etc., and check for termination
}
```

Alternatively, instead of sending tags we can send executable messages, such as Java runnable objects, C++ function objects or lambdas, C# delegates or lambda expressions, or even C function pointers. This helps simplify the worker thread mainline:

```
// Option 1(b): Queueing runnable work for a dedicated
// background thread. Suitable where tasks need
// to run serially with respect to each other (note
// corollary: an earlier task can block later tasks).
//
// GUI thread
while( message = queue.Receive() ) { // this could block
    if( it's a "save document" request ) {
```

```

TurnSavingIconOn();
worker.Send( [] { SaveDocument(); } ); // send async work
}
else if( it's a "print document" request ) {
    TurnPrintingStatusOn();
    worker.Send( [] { PrintDocument(); } ); // send async work
}
else if( it's a "save document" notification ) { ... }
// as before
else if( it's a "print document" progress notification ) { ... }
// as before
else
    ...
}

// Simplified dedicated worker thread:
// Just executes the work it's being given.
//
while( message = workqueue.Receive() ) { // this could block
    message(); // execute the given
    // work
    ... // check for
    // termination
}

```

Async Option 2: Background Thread Per Task

Our second option is a variant of the first: Instead of having only one dedicated background worker, we can choose to launch each piece of asynchronous work as its own new thread. This makes sense when the work items really are independent and won't interfere with each other, though it can mean that work that is launched later can finish earlier, as illustrated in Figure 2.

Here's sample code that sketches how we can write this option, where red code again highlights the code that is different from the previous example.

```

// Option 2: Launching a new background thread
// for each task. Suitable where tasks don't need
// to run sequentially with respect to each other.
//
while( message = queue.Receive() ) { // this could block
    if( it's a "save document" request ) {
        TurnSavingIconOn();
        ... new Thread( [] { SaveDocument(); } ); // run async request
    }
    else if( it's a "print document" request ) {
        TurnPrintingStatusOn();
        ... new Thread( [] { PrintDocument(); } ); // run async request
    }
    else if( it's a "save document" notification ) { ... }
    // as before
    else if( it's a "print document" progress notification ) { ... }
    // as before
    else
        ...
}

```

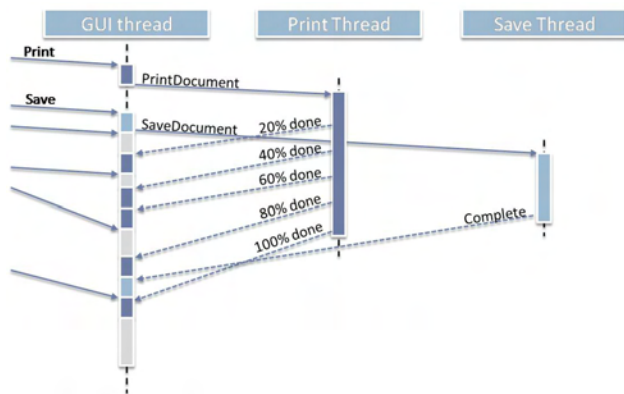


Figure 2: Getting work off the GUI thread using separate background workers.

The ellipsis before *new Thread* stands for any housekeeping we might do to keep track of the threads. In languages with garbage collection we would normally just launch a *new Thread* and let it go; otherwise, without garbage collection we might maintain a list of launched threads and clean them up periodically.

Async Option 3: Run One-Shot Independent Tasks On a Thread Pool

Thread pools are about expressing independent work that will get run on a set of threads whose number is automatically chosen to match the number of cores available on the machine. Pools are mainly intended to enable scalability (Pillar 2), but can sometimes also be used for running what would otherwise be short threads.

Figure 3 shows how a short-running thread that rarely waits/blocks can instead “rent-a-thread” to run as a pool work item. Again, this technique is for simple one-shot work only, and it's very important not to run work on a thread pool if the work could block, such as wait to acquire a mutex or wait to receive a message (sending out messages is okay because that doesn't block).

Here's some sample code, which again should be considered pseudocode given that the specific spelling of “run this work on that pool” varies from one language and operating system to another:

```

// Option 3: Launching each task in a thread pool.
// Suitable for one-shot independent tasks that
// don't block for locks or communication.
//
while( message = queue.Receive() ) { // this could block
    if( it's a "save document" request ) {
        TurnSavingIconOn();
        pool.run( [] { SaveDocument(); } ); // async call
    }
    else if( it's a "print document" request ) {
        TurnPrintingStatusOn();
        pool.run( [] { PrintDocument(); } ); // async call
    }
    else if( it's a "save document" notification ) { ... }
    // as before
    else if( it's a "print document" progress notification ) { ... }
    // as before
    else
        ...
}

```

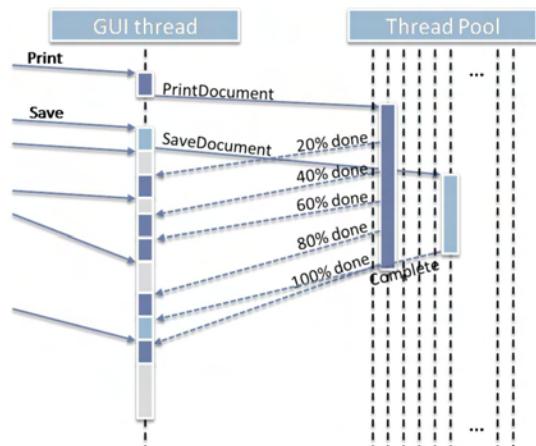


Figure 3: Getting work off the GUI thread using a thread pool (appropriate for shorter and non-blocking work).

A Word About OpenMP and Work-Stealing Runtimes

Table 1 shows two other tools that are available (or becoming available) to express potentially asynchronous work, but for a different purpose:

- Work-stealing runtimes. Work stealing, pioneered by Cilk [3], is the engine behind the emerging crop of next-generation runtimes that target providing scalable concurrency (Pillar 2). They make it easy and efficient to express work that might be done in parallel if there are enough hardware resources to run it in parallel, yet add hardly any overhead if there aren't extra cores handy because the default implementation is to run the work sequentially in the original thread. Products based on work stealing include Intel's Threading Building Blocks [4], and at least four upcoming products: the Visual C++ 2010 Parallel Patterns Library (PPL) [5], the .NET 4.0 Task Parallel Library (TPL) [6] and Parallel LINQ (PLINQ) [7], and the Java 7 Fork/Join framework [8].
- OpenMP. Long available and now in version 3.0, OpenMP is all about parallelizing Fortran- and C-style loops, running them on a sort of thread pool under the covers to get data-parallel scalability. [9]

I mention these tools here primarily to say that they're meant for Pillar 2, as Table 1 shows. They're all about different ways to split and subdivide work across available cores to get the answer faster on machines having more cores, from doing loop iterations in parallel, to working on subranges of the data in parallel, to performing recursive decomposition (divide-and-conquer algorithms) in parallel.

These are not the right tools for Pillar 1, which means that they neither compete with threads nor replace them. The reason that work-stealing runtimes and OpenMP are unsuitable for running work asynchronously is because the work is not guaranteed to actually run off the original thread—i.e., it's not guaranteed to be asynchronous at all.

For completeness, though, here's a taste of the syntax:

```
// Option 4 (NOT recommended for PrintDocument):
// Run on a work stealing or OpenMP runtime.

// Using Visual C++ 2010 Parallel Patterns Library (PPL)
// with the convenience of ISO C++0x lambdas
//
taskgroup.run( [] { PrintDocument(); } );

// Using .NET 4.0 Task Parallel Library (TPL)
// with the convenience of C# lambdas
//
Parallel.Invoke( () => { PrintDocument(); } );

// Using Java 7 ForkJoin with an explicit runnable object
//
class PrintTask extends ForkJoinTask {
    public void run() { PrintDocument(); }
}
...
Fjpool.execute( new PrintTask(); );
```

OpenMP is not based on work stealing, but does offer a similar construct with the same caveat:

```
// Using OpenMP (again, NOT recommended for PrintDocument)
//
#pragma omp task
{
    PrintDocument();
}
```

Example 2: Sockets

Another classic example of event-driven code is client-server communications. A server listens for incoming connection requests; as each connection arrives, the server has to accept it and then process inbound and outbound traffic on that connection until the connection is closed by one side or the other. At any given time, the server has to be able to deal with multiple active connections from different clients, each at a different stage in its processing.

As with the previous GUI example, we have the option of handling the work synchronously or asynchronously. The synchronous code looks something like this in a classic socket-oriented API. The methods are spelled a little differently in POSIX, Java, and other environments, but all follow the basic operation.

```
// Example 2: Doing it all on one thread/process
// (pseudocode, and assumes select returns single events)
//
while( /*...*/ ) {
    select( our socket list ); // blocks until something is ready

    if( a connection request came in )
        accept() and add it into our socket list

    else if( a connection closed )
        close() and remove it from our socket list

    else if( one of our read sockets is ready )
        // synchronous processing
        read more data from that socket and process it

    else if( one of our write sockets is ready )
        // synchronous processing
        write the next chunk of data to that socket
}
```

A key problem with this is that all the work is being handled by a single thread. For one thing, that tends to make the code more complex; one thread has to do it all, so all the code to handle all the different kinds of clients appears in one place. For another, running all the work on one thread means that all the work will run sequentially on one core (or, equivalently, one core at a time if the operating system migrates the thread from core to core). If the total work saturates the core, then additional work will be throttled and slow down the server's responsiveness for all clients as later requests queue up and wait for earlier work to complete.

A common solution is to use Option 2: Let the server service connections simultaneously by launching a new thread to handle each incoming client. Here's the basic logic:

```
// Applying Option 2 (pseudocode)

// Listener thread: Only accepts connections
// and launches new threads to deal with them
//
```

```

while( /*...*/ ) {
    wait for connection request
    accept()
    ... = new Thread( [] { HandleIt( theConnection ); } );
}

// Each handler thread
//
void HandleIt( ... ) {
    while( wait for event on this connection ) {
        // this could block
        if( this connection was closed ) {
            close();
            return;
        }
        else if( incoming data ready to read )
            // synchronous processing
            read more data from this connection and process it
        else if( connection is ready to write )
            // synchronous processing
            write the next chunk of data to this connection
    }
}

```

This code is both clearer and potentially faster. The program is clearer because the code to handle each kind of client is nicely wrapped up in its own method or class, instead of being intermingled. The program is faster because it keeps different connections asynchronous and independent, so that the work on one connection doesn't have to wait to be processed sequentially behind work on another connection. This gives better responsiveness even on a single-core server that isn't heavily loaded, and it delivers better scalability under load on servers that do have parallel hardware.

Example 3: Pipeline Stages

A third classic example of independent work that should run asynchronously is pipelining, and the independent pieces of work are the pipeline stages. In a nutshell, here's the idea:

We have a series of pieces of data to be processed in order (e.g., series of packets to prepare for sending).

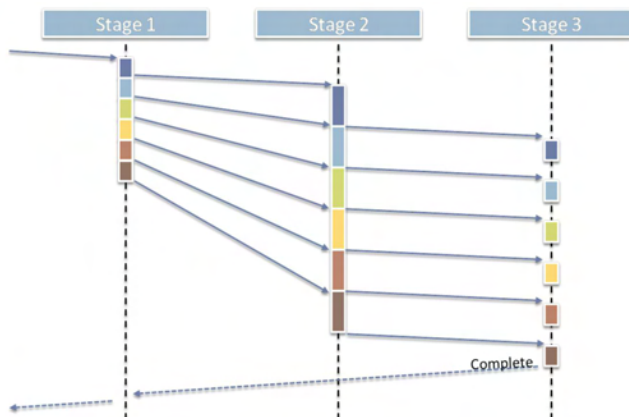


Figure 4: Using three threads to express three pipeline stages

Each piece has to go through several stages of processing (e.g., decorate, compress, encrypt) that need to be applied to a given piece of data in order.

The stages are otherwise independent (e.g., the compressor can run independently of the decorator and encryptor), other than waiting for available work to arrive from the previous stage.

As shown in Figure 4, we can express the pipeline with one thread per stage, connected by asynchronous message queues to let the stages run independently and tolerate latency as the different stages typically run at different speeds. Each stage just takes each incoming packet from the previous stage, does its own processing, and throws it over to the next stage.

Summary

Threads are a low-level tool for expressing asynchronous work. “Up-level” them by applying discipline: strive to make their data private, and have them communicate and synchronize using asynchronous messages. Each thread that needs to get information from other threads or from people should have a message queue, whether a simple FIFO queue or a priority queue, and organize its work around an event-driven message pump mainline; replacing spaghetti with event-driven logic is a great way to improve the clarity and determinism of your code.

Notes

- [1] H. Sutter. “The Pillars of Concurrency” (*Dr. Dobb's Journal*, August 2007). Available online at <http://www.ddj.com/architect/200001985>.
- [2] E. Lee. “The Problem with Threads” (*EECS Department Technical Report*, University of California, Berkeley, 2006). Available online at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>.
- [3] The Cilk Project website: <http://supertech.csail.mit.edu/cilk/>.
- [4] Intel Threading Building Blocks website: <http://www.threading-buildingblocks.org/>.
- [5] K. Kerr. “Visual C++ 2010 and the Parallel Patterns Library” (*MSDN Magazine*, February 2009). Available online at <http://msdn.microsoft.com/en-us/magazine/dd434652.aspx>.
- [6] D. Leijen and J. Hall. “Optimize Managed Code For Multi-Core Machines” (*MSDN Magazine*, October 2007). Available online at <http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>.
- [7] J. Duffy and E. Essey. “Running Queries On Multi-Core Processors” (*MSDN Magazine*, October 2007).
- [8] D. Lea. “Package jsr166y” (Draft JSR166y documentation). Available online at <http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/>.
- [9] OpenMP Application Program Interface Version 3.0 (OpenMP.org, May 2008). Available online at <http://www.openmp.org/mp-documents/spec30.pdf>.