United Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development December, 2009

Bilski and the Problem with Software Patents

By Jonathan Erickson,
Editor In Chief

On the off chance you haven't been following what's known as the "Bilski case" — well, you should be following it. Of course, at this stage, it's more "catch up" than "follow" since the case is now in the hands of the U.S. Supreme Court and nothing will likely happen until the middle of next year when the Court lays down the law, so to speak.

If it seems that we've been wrestling with the issue of software patents forever, you're right. Maybe, just maybe, we'll have some direction in the coming year.

Here's the background: Bernard Bilski and Rand Warsaw applied for a U.S. patent for "a method of hedging risk in the field of commodities trading." The application was rejected by both the U.S. Patent and Trademark Office (USPTO) and the patent appeals court, which found that the application was not subject matter that qualifies for patent protection. Bilski then appealed the rejection to the Federal Circuit, and again was rejected. Finally, at the start of this year, Bilski and Warsaw petitioned the U.S. Supreme Court to overturn the Federal Circuit's decision, and oral arguments were presented to the Justices last week. And so we now wait to hear their decision.

So why should software developers care about a twice-denied patent application for something that has to do with commodities trading? The fundamental issue, says Robert Tosti, an attorney specializing in patents, trademarks, and other such intellectual property, is whether patents should be allowed on methods that don't involve a machine. Stuff like, well, source code. In other words, the question of whether software can be protected by patents — an issue that's been bubbling for 20 years or more — may finally be laid to rest.

Both the Free Software Foundation and Red Hat submitted amicus curiae briefs asking that the Supreme Court to affirm that software is not patentable, with Red Hat pointing out that "our patent system is supposed to foster innovation, but for open source and software in general, it does the opposite. Software patents form a minefield that slows and discourages software innovation." Hear, hear!!

In the meantime, we sit and wait a little while longer. The one thing we can count on when the Justices render their opinion is that some of us will like what they say, and some of us won't.

Return to Table of Contents

The Discrete Fourier Transform

The theories of an early-19th-century French mathematician have emerged from obscurity to become part of the basic language of engineering

by Larry Hardesty

In 1811, Joseph Fourier, the 43-year-old prefect of the French district of Isere, entered a competition in heat research sponsored by the French Academy of Sciences. The paper he submitted described a novel analytical technique that we today call the Fourier transform, and it won the competition; but the prize jury declined to publish it, criticizing the sloppiness of Fourier's reasoning. According to Jean-Pierre Kahane, a French mathematician and current member of the academy, as late as the early 1970s, Fourier's name still didn't turn up in the major French encyclopedia the *Encyclopdia Universalis*.

Now, however, his name is everywhere. The Fourier transform is a way to decompose a signal into its constituent frequencies, and versions of it are used to generate and filter cellphone and Wi-Fi transmissions, to compress audio, image, and video files so that they take up less bandwidth, and to solve differential equations, among other things. It's so ubiquitous that "you don't really study the Fourier transform for what it is," says Laurent Demanet, an assistant professor of applied mathematics at MIT. "You take a class in signal processing, and there it is. You don't have any choice."

The Fourier transform comes in three varieties: the plain old Fourier transform, the Fourier series, and the discrete Fourier transform. But it's the discrete Fourier transform (DFT) that accounts for the Fourier revival. In 1965, the computer scientists James Cooley and John Tukey described an algorithm called the fast Fourier transform, which made it much easier to calculate DFTs on a computer. All of a sudden, the DFT became a practical way to process digital signals.

To get a sense of what the DFT does, consider an MP3 player plugged into a loudspeaker. The MP3 player sends the speaker audio information as fluctuations in the voltage of an electrical signal. Those fluctuations cause the speaker drum to vibrate, which in turn causes air particles to move, producing sound.

An audio signal's fluctuations over time can be depicted as a graph: the x-axis is time, and the y-axis is the voltage of the electrical signal, or perhaps the movement of the speaker drum or air particles. Either way, the signal ends up looking like an erratic wavelike squiggle. But when you listen to the sound produced from that squiggle, you can clearly distinguish all the instruments in a symphony orchestra, playing discrete notes at the same time.

That's because the erratic squiggle is, effectively, the sum of a number of much more regular squiggles, which represent different frequencies of sound. "Frequency" just means the rate at which air molecules go back and forth, or a voltage fluctuates, and it can be represented as the rate at which a regular squiggle goes up and down. When you add two frequencies together, the resulting squiggle goes up where both the component frequencies go up, goes down where they both go down, and does something in between where they're going in different directions.

The DFT does mathematically what the human ear does physically: decompose a signal into its component frequencies. Unlike the analog signal from, say, a record player, the digital signal from an MP3 player is just a series of numbers, representing very short samples of a real-world sound: CD-quality digital audio recording, for instance, collects 44,100 samples a second. If you extract some

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

VICE PRESIDENT, GROUP PUBLISHER
Brandon Friesen
VICE PRESIDENT GROUP SALES
Martha Schwartz

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

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

UBM LLC

Pat Nohilly Senior Vice President, Strategic Development and Business Administration
Marie Myers Senior Vice President, Manufacturing

TechWeb

Tony L. Uphoff Chief Executive Officer
John Dennehy, CFO
David Michael, CIO
John Siefert, Senior Vice President and Publisher, InformationWeek Business Technology Network
Bob Evans Senior Vice President and Content Director, InformationWeek Global CIO
Joseph Braue Senior Vice President, Light Reading Communications Network
Scott Vaughan Vice President, Marketing Services
John Ecke Vice President, Financial Technology Network
Beth Rivera Vice President, Human Resources
Fritz Nelson Executive Producer, TechWeb TV

techweb™

number of consecutive values from a digital signal — 8, or 128, or 1,000 — the DFT represents them as the weighted sum of an equivalent number of frequencies. ("Weighted" just means that some of the frequencies count more than others toward the total.)

The application of the DFT to wireless technologies is fairly straightforward: the ability to break a signal into its constituent frequencies lets cellphone towers, for instance, disentangle transmissions from different users, allowing more of them to share the air.

The application to data compression is less intuitive. But if you extract an 8x8 block of pixels from an image, each row or column is simply a sequence of eight numbers — like a digital signal with eight samples. The whole block can thus be represented as the weighted sum of 64 frequencies. If there's little variation in color across the block, the weights of most of those frequencies will be zero or near zero. Throwing out the frequencies with low weights allows the block to be represented with fewer bits but little loss of fidelity.

Demanet points out that the DFT has plenty of other applications, in areas like spectroscopy, magnetic resonance imaging, and quantum computing. But ultimately, he says, "It's hard to explain what sort of impact Fourier's had," because the Fourier transform is such a fundamental concept that by now, "it's part of the language."

Return to Table of Contents

The SEMAT Initiative: A Call for Action

by Ivar Jacobson, Bertrand Meyer and Richard Soley

As you may have heard, we have been quietly planning a revolution with the goal of "re-founding" software engineering as a rigorous discipline. We recognize that the natural tendency in our field is to perturb systems minimally into approximate correctness, but this path cannot be sustained any longer if we are to support the computing industry and help it meet the demands of society. We need to restart on a solid basis, taking advantage of all that has been learned in software engineering theory and practice over the past five decades.

The initiative started with two articles published in *Dr Dobb's* — "Methods Need Theory" (www.ddj.com/architect/219100242)> and "Why We Need a Theory for Software Engineering" (www.ddj.com/architect/220300840) — that analyzed and deplored the fragmentation of software engineering and, in particular, of the methodology scene.

To address this issue, to bring together the best and the brightest, we have founded a community, Software Engineering Method and Theory (SEMAT) community (www.semat.org). The community supports the following "Call for Action."

SEMAT Call for Action Statement

Software engineering is gravely hampered today by immature practices. Specific problems include:

- The prevalence of fads more typical of fashion industry than of an engineering discipline.
- The lack of a sound, widely accepted theoretical basis.
- The huge number of methods and method variants, with differences little understood and artificially magnified.
- The lack of credible experimental evaluation and validation.
- The split between industry practice and academic research.

We support a process to refound software engineering based on a solid theory, proven principles, and best practices that:

- Includes a kernel of widely-agreed elements, extensible for specific uses.
- Addresses both technology and people issues.
- Is supported by industry, academia, researchers and users.
- Supports extension in the face of changing requirements and technology.

Signatories

The following people have signed the Call for Action statement. This list is dynamic; the up-to-date list is on the SEMAT web site (www.semat.org).

Pekka Abrahamsson
Scott Ambler
Victor Basili
Jean Bézivin
Dines Bjorner
Barry Boehm
Alan W. Brown
Alistair Cockburn
Larry Constantine
Bill Curtis
Donald Firesmith
Erich Gamma
Tom Gilb
Ellen Gottesdiener
Sam Guckenheimer
David Harel
Brian Henderson-Sellers
Watts Humphrey
Capers Jones
Martin Griss
Ivar Jacobson
Philippe Kruchten
Robert Martin
Stephen Mellor
Bertrand Meyer
James Odell
Meilir Page-Jones
Ken Schwaber
Alec Sharp
Richard Soley

All signatories are world-class experts who have individually made significant contributions to the software engineering discipline and of the development of today's best practices. They are distributed around the world; their collective expertise extends across computer science and software engineering, covering advanced programming techniques, modeling languages, Agile as well as traditional software processes, a wide spectrum of practices (human, technical, business models, user interaction), tools, frameworks, and many more areas. Together we represent a large part of the software development community.

Questions and Answers

Q: The goals are lofty, but they are only goals. Before going public, shouldn't you have waited until you had some actual solutions to show?

A: Lots of people have excellent ideas, but no one holds the whole truth. The only way to solve a problem of this magnitude is through a community effort that identifies the problem precisely and leads to widely agreed solutions. That's why we are starting with a statement of objectives and encouraging comments from a wide range of experts.

Q: Why the emphasis on signatories?

A: The people who have agreed to sign the Call for Action have, each in his or her own way, had an effect on the world of software development. Their participation is essential to the visibility of the project, in particular with large companies, and to its success.

Q: There is little visible consensus among these signatories. How will you get them to agree on a common approach?

A: By its very nature this is indeed a group made of people with strong views. But they realize the importance of coming to a kernel of widely-agreed elements, solid enough to address the core problems and extensible enough to accommodate the diversity of requirements, usage contexts and technologies. The first step is to develop a joint vision statement that will serve as a blueprint for the rest of the effort.

Q: Now that there's a Call for Action signed by an impressive group of people, will the real work start?

A: Yes, the hard part lies ahead. The hardest part will be to get everyone moving forward without compromising ourselves to death, and without anyone giving up because his or her pet idea is not the key driver. We think that with perseverance, creativity and diplomacy we will be successful.

Q: There are plenty of approaches to software engineering discipline and methodologies. We don't need a revolution, we need wide acceptance of my theory!

A: A number of people have superb theories, sometimes backed by successful experiments; no doubt yours is one of them. Unfortunately, no single approach has garnered broad acceptance in industry and academia. Like other engineering fields, our's needs a common framework with which everyone agrees. We hope you will contribute your ideas.

Q: How is SEMAT different from other earlier efforts, such as, the IEEE Computer Society SWEBOK (Software Engineering Body of Knowledge) or the various standard curricula in software engineering?

A: A crucial difference in the SEMAT initiative from previous efforts, such as SWEBOK, is the central focus on underlying theory. While SWEBOK and the various IEEE curricula reflect the state of the art, those constitute purely pragmatic admixtures of ideas, constructs, techniques, and disciplines along with current practice and popular opinion. SEMAT is focused on a kernel of strong theoretical underpinnings. That's not to say that SWEBOK and the like aren't valuable, and indeed valuable inputs to the SEMAT initiative. Only that we need a theory!

In the long run, software engineering needs to become grounded in theory and based on evidence, criteria met by only a small fraction of what passes for software engineering as taught and as practiced today. In order to rise to some such standard of evidence and theory, SEMAT will need to be winnowing down rather than gathering in. While this work will certainly draw on SWEBOK and other recognized sources, rather than expanding and including, as must be the agenda for a body of knowledge or a curriculum, much effort may have to be in narrowing and excluding. In the process, it is hoped to identify precisely some of those areas in most critical need of attention, either to test and prove theory or to build explanation that encapsulates evidence and experience.

Q: The call for action talks about "a kernel of widely-agreed elements, extensible for specific uses". This sounds very much like a unified methodology. Is it?

A: No, we are not looking for a single methodology. But we do seek to identify "universals" of software development: universal problems and universally recognized solution elements. For example, every project has a process and products, and these products must always include, in the end, executables. Every project must do some form of analysis, of design, of implementation, of verification. Other universals include roles: analyst, developer, QA engineer. The SEMAT effort should identify these universals, both in

the problem space and in the solution space. This will make it easier not only to recognize solved problems and reuse recognized solutions (theories and practices), but also to compose solutions and, where none exist, develop new ones.

Q: Why would we succeed where so many have failed?

A: We believe that the industry is tired of fads, wary of empty talk about methods and process, and ready for a change We also see that the chasm between industry and academia is being questioned, with (for example) many practices that originated in industry having found their way into standard university curricula, and some advanced research finding its way into actual projects (initially in mission-critical systems, but with potential for much wider spreading).

For these hopes to be realized, the community -- industry and research -- must join forces. While the amount of work that remains is huge, the momentum already created by the SEMAT initiative, over the course of just a few weeks, shows that this prospect is achievable. We hope you will share our enthusiasm.

Do You Want To Support Us?

Take a look at www.semat.org. But you can do more than visiting! First, you can become a supporter by clicking "sign up" on the site. You can help with ideas for the vision statement on which we are now working. You can contribute to the blog on the site, and participate in the discussions. Right now, you can spread the word. This initiative needs to be widely supported if it is going to have any impact on the industry. We need to do a good job, but when we don't you can correct us. As the vision statement and further material becomes available, you can help develop training material for industry and universities. There will be many more ways to participate.

What Are Our Success Criteria

To be successful the SEMAT effort will require:

- Broad acceptance of an agreed kernel, even an initially small one, combining existing ideas and possibly new ones to define a rigorous basis for the industry.
- >Recognition of that kernel in academic and research communities, so that there is training, education, books and other materials to educate the profession.
- Acceptance of that kernel in industrial settings, i.e. in the development of new software.

The longer we wait, the longer we perpetuate the impression by the rest of the world that software is too often brittle. Let's stop wasting time, and get to work.

—Ivar Jacobson is a father of components and component architecture, use cases, aspect-oriented software development, modern business engineering, the Unified Modeling Language and the Rational Unified Process. Bertrand Meyer is one of the pioneers of object technology and the designer of the Eiffel method and language. He is Chief Architect of Eiffel Software and Professor of Software Engineering at ETH Zurich. Richard Soley is chairman and CEO of the Object Management Group, which developed the OMG standards-setting process and the original CORBA specification.

Return to Table of Contents

Secure Software Needs Careful Testing — And Lots Of It

With fuzzing, we deliberately attack software with random data in search of unexpected responses

by Herbert H.Thompson

Software testing comes in many flavors. Unit testing analyzes individual components before they're integrated into larger systems. System and integration testing checks that modules work together. Regression testing verifies that everything still works after a change is made to the code. And security testing checks that data is protected.

Tools such as source-code scanners, security-aware compilers, and application scanners help developers find vulnerabilities in code. And techniques like fuzz testing uncover inputs that can cause apps to behave badly.

With fuzzing, we deliberately attack software with random data in search of weaknesses and unexpected responses. Fuzz testing is particularly important in Web application development, and it's playing a growing role in ensuring that "security quality" — the confidentiality, integrity, and availability of systems and data for users — is integrated into every phase of development.

Fuzz testing finds vulnerabilities that are too complicated for a person to see. With complex programs — Windows 7 has an estimated 50 million lines of code, for instance — it's difficult to anticipate everything that can go wrong. Fuzzing takes a "big hammer" approach to systems, hitting them persistently with strange inputs. Fuzzers also save companies money because they don't require much human intervention. Microsoft fuzzes its apps until end of life because it's so cheap to do. Another advantage of fuzz testing is its simplicity. Tests that target specific vulnerabilities can quickly be implemented. Fuzzing is credited recently with finding vulnerabilities in the Windows 7, iPhone, and Android communications protocols.

Early Fuzzing

My first fuzzing experience was in the late 1980s when soda vending machines appeared in the Bahamas, where I grew up. But the machines had signs on them saying "U.S. Quarters Only." The Bahamian currency is pegged one-to-one with the U.S. dollar, and both currencies are accepted everywhere. When my friends and I came across our first soda machine, none of us had a U.S. quarter, so we started experimenting. One of us put a washer in the slot. It dropped right through. We tried a U.S. dime. Nothing. A Bahamian quarter. Still nothing. The outlook for soda was bleak until we dropped a Bahamian 10-cent piece in.

To our amazement, the display instantly registered "25" — the machine mistook the 10-cent coin for a U.S. quarter. We scrambled to find three more 10-cent pieces. The machine read "50," "75," and finally "1.00." We hit Select and got a soda.

Years later, I learned that the vending machine measured diameter and weight to determine if an object dropped into the slot was a U.S. quarter. Bahamian 10-cent pieces coincidently have almost exactly the same diameter and weight as U.S. quarters.

Without understanding the mechanics of the machine, we had applied semi-random inputs and

hoped for something interesting to happen. What we were doing is now known as "fuzzing" — applying inputs with some degree of randomness, then looking for unexpected outputs. In theory, it sounds straightforward, but in practice, it can be difficult to do well.

How Fuzzing Works

There are many freely available fuzzers, such as Peach Fuzzer from Eddington and Frantz, Spike from Immunity, and MiniFuzz from Microsoft, as well as commercial fuzzers such as Codenomicon from Codenomicon Defensics and Beyond Security's BeStorm. Web application fuzzers are by far the most mature. They understand how an application is supposed to respond to normal input, then they either look for deviations or symptoms of failure. When testing for SQL Injection vulnerabilities, for instance, an input might be a string of text with a single quote mark. If a database error message is returned, the tester knows there's a potential vulnerability. For non-Web software, fuzzing is murkier. These fuzzers work by corrupting files, network traffic, or API parameters by doing the following:

1. They generate a random input (or randomly corrupt an existing input).
2. Deliver that input to the app.
3. Monitor it for an exception or crash.

This approach finds problems such as buffer overflows, a symptom of which is usually a crash. If you're looking for other, non-crashing vulnerabilities, fuzzing can still be useful if it can be automated.

Back to my soda machine, we were looking for one thing — the red display showing the money we put in had registered. In retrospect, there were other outcomes that would have been interesting: The machine might have crashed, short-circuited, or become hopelessly jammed. If you can specify the symptoms you're interested in, fuzzing can be an incredibly cheap way to see how a system responds to a large set of inputs. The problem is that it can miss more subtle architectural problems.

Beyond the Requirements

Most security vulnerabilities aren't requirements violations; rather, they come from incomplete requirements. For example, a simple requirement takes the form of "when a user applies input x, the software should produce output y." It's easy to test such requirements: Apply input x, then look for y.

Verifying requirements is critical, but security testing looks beyond intended behaviors set out in requirements. Behind those may be a lot of unspecified assumptions and constraints. For example, if the input or output is sensitive (say, a credit card number), storing it even temporarily in an insecure place is a bad idea. Developers may overlook this and make design and implementation choices that leave the software at risk. To instill security quality into a project, testers use fuzzing and similar techniques to look beyond stated requirements and get answers to questions like:

- What isn't the system supposed to do?
- How should inputs, functionality, and data be restricted?
- Are security features correct and is functional code secure?

Vulnerabilities can be introduced at any stage of development. A key security requirement might be poorly defined or missing. An architectural weakness could get introduced during design. Developers may use a vulnerable function to process user input. Security testing tools can check for these vulnerabilities, but there are limitations.

Over the past few years, several classes of security testing tools have emerged. Essentially, these tools, some of which incorporate fuzz testing, have taken over the security aspects of testing. While they tend to find certain classes of vulnerabilities, they miss many others. They aren't substitutes for security testing, but instead, like a pair of pliers, they extend your reach when testing software. Bottom line, it's people thinking creatively about how to misuse or abuse software who drive security testing. For example, a source-code scanner or security-aware compiler can look for functions and constructs that commonly result in vulnerabilities. They can alert you that the code may have a weakness even when the syntax is essentially correct. While these tools can be useful during development and testing, they see only part of the picture.

Most applications are highly interconnected and have lots of interaction with other software. Statically scanning the code of one application or module doesn't give a complete view of how a running application will respond to hostile input. Some dynamic security tools exist to help bridge the gap. Application scanners primarily focusing on Web apps create input strings that can reveal potential vulnerabilities; they also look for symptoms of failure. These tools can be good at finding low-hanging fruit and tend to focus on some of the more common vulnerabilities.

Architectural Vulnerabilities

We were about 13 sodas into our discount scheme when we discovered another design choice that would make the soda machine flaw far worse — when the machine ran out of sodas and a little red "Empty" light came on. Not willing to take a loss on the 40 cents we'd just invested, we pressed the "Coin Return" button. We assumed the machine would give us back the four Bahamian 10-cent pieces. However, pressing the button returned four U.S. quarters — a 150% return on our investment! This was a definite upgrade over free sodas.

On its own, the coin-return design was likely a good one based on mechanics and convenience. The developer relied on the assumption that the rest of the system worked as intended, including that it correctly identified objects as quarters. The coin return design choice by itself wasn't a security vulnerability, but it severely increased the impact of the machine's existing problem, mixing up U.S. quarters and Bahamian 10-cent pieces. To have a good approach to systems security, you have to practice the principle of defense-in-depth, which forces design choices that create a safety net around potential problems. Problems like this speak to the need of a more holistic approach to security testing.

Software security quality should be woven throughout the development process. It begins in requirements and design, is propagated through development and testing, and continues into deployment and support. The good news is that there are lots of resources. Microsoft has made many of its processes and tools available from its Security Development Lifecycle, a process that provides security and privacy throughout the development process. Likewise, SAFECode is synthesizing and distributing security best practices from some of the world's largest software makers.

The various secure development methodologies have some common themes, with the need for education perhaps being the biggest. The term "education" isn't exactly right — what's really needed is re-education so developers understand that security isn't a natural outcome of traditional quality-assurance processes. Building secure software takes focused work and a different mindset. In a few years, if all goes well, perhaps we'll be beyond worrying about security quality because security will be woven into software in the same way that reliability is. But until that happens, I still have a few more sodas to buy.

— Herbert H. Thompson is chief security strategist for People Security, a security education company. He also teaches software security at Columbia University in New York.

Return to Table of Contents

Is Larrabee For the Rest of Us?

Can non-numerical application developers take advantage from the new LRBni instructions?

by Daniele Paolo Scarpazza

Regardless of whether it will see market in 2010, Larrabee (the many-core architecture that Intel has been developing since the spring of 2007) is going to catch most software designers unprepared. The details available now leave fundamental questions unanswered: What is Larrabee's potential performance? Which programming model should we adopt to utilize the processor fully? These challenges are even harder for programmers of non-numerical applications, traditionally considered a secondary audience by designers of graphics-oriented architectures. With Intel ditching its plans for Larrabee-based discrete graphics products only days ago, and refocusing Larrabee primarily toward the High-Performance Computing (HPC) community, these questions become crucial.

In this article, I'll use the example of regular-expression matching (arguably one of the hardest non-numerical workloads to parallelize) and map it onto LRBni, Larrabee's new instruction set. I'll show that a good mapping of this code onto LRBni is possible, but only at the cost of a radical data-parallel redesign. Indeed, the data-parallel techniques that I learned on earlier SIMD (Single Instruction, Multiple Data) architectures are even more effective on Larrabee. Because of the wide SIMD nature of Larrabee, developers who don't adopt a similar data-parallel approach risk leaving most of the processor's compute power on the table.

At this point, nobody outside Intel can estimate how fast Larrabee will run the code I provide. With such complex instructions, some of which translate into multiple micro-instructions and potentially involve multiple cache misses, it will be a challenge for Intel engineers to match the throughput and power efficiency of GPGPUs and more streamlined RISC architectures.

Why Regular Expressions?

Regular expressions (RE) are those patterns that you feed to grep on the command line when searching log files for messages matching a given template, the ones that you provide to flex when designing the tokenizer for your newly invented programming language, the ones that tell network intrusion detection systems like Snort what malicious traffic looks like, etc.

You might wonder why, among many possible workloads I could examine, I chose regular expression matching. Here are five good reasons:

- RE matching is the core of common and emerging applications like search-engine indexing, XML processing, application-level intrusion detection, and language analytics. In all of these, it is one important (if not the most important) consumer of execution time and hardware resources.
- RE matching relies on finite-state machines, a class of workloads notoriously hard to parallelize. In his famous presentation [4] — almost a manifesto — on parallel computing research, David Patterson mentions finite-state machines (FSM) as the #1 of his "dwarfs", i.e., workloads that deserve attention and effort.
- Larrabee is a SIMD architecture. On a SIMD architecture, each register contains multiple operands, and a SIMD instruction can process all those operands in parallel. We are used to 128-bit registers, divided in four 32-bit lanes. Larrabee has 512-bit registers, divided in 16 lanes. Traditionally, compilers don't usually do a very good job in translating irregular, non-numerical code into SIMD code. If you want good code, chances are you have to write SIMD code yourself, by hand. If you don't, you might leave up to 75% of the performance on the table on contemporary 128-bit SIMD architecture. On Larrabee, the potential loss is even higher: You might be using only 1/16 of the compute power (i.e., waste 93.75%).

- Presenting a realistic application in depth in this short space (and within the short attention span of a busy programmer) is a challenge. RE matching boils down to a handful of instructions, making this possible.
- RE matching is not your usual GPU workload. GPU architects catered primarily to the gaming and the computer graphics communities. More recently, the scientific and supercomputing communities have gained some attention by showing that GPGPUs can accelerate numerical, dense algorithms from domains such as fluid dynamics, protein folding, and acoustics. But "the rest of us" — developers of non-numerical applications — are still at the door, waiting. I use this example to show a way to exploit GPGPUs to carry out jobs previously thought "hard" to parallelize. There is no reason why, in the near future, most desktop applications (and not just games or graphics codes) should not offload their heavyweight kernels to GPGPUs.

The main message of this article is that the programming challenges that LRBni brings about are not trivial, but also not new, and that they can be attacked successfully with data-parallel techniques proven effective in the past. I propose an approach based on the complete elimination of the control flow from the compute kernels (the hot spots of your application), and its transformation into data flow.

Thinking in terms of branch-less data flows is the most natural way to take advantage of SIMD. In the past, I was able to apply this approach on the same RE matching workload I am presenting here, while designing a SIMD-based tokenizer kernel [5] intended for use in a search engine indexer [6] on the IBM Cell Broadband Engine processor. This kernel achieved 99.3% of cycle utilization, is the fastest software tokenizer available on any architecture, and delivers, with some restrictions, a level of throughput comparable to dedicated hardware.

Much Ado About Something?

Larrabee has been constantly surrounded by hype, mystery and controversy. More than a year has passed since it was first described in a SIGGRAPH 2008 paper entitled "Larrabee: A Many-Core x86 Architecture for Visual Computing" (http://portal.acm.org/citation.cfm?doid=1360612.1360617), and nobody outside Intel seems to have seen silicon of it (with the exception of Intel's brief demo at SuperComputing 2009 in November. The anticipation keeps growing. While Intel keeps parceling out details and competitors react with anxiety and criticism, we developers are left wondering if it makes sense to commit resources and attention to the platform in the hope of accelerating most applications, or if Larrabee will just work well with regular, numeric, array-based workloads (gaming graphics and physics) and won't be a true game-changer for everyone else.

A stream of press releases, rumors, and blog posts have created more confusion. Less than two weeks ago, Intel announced that its plans to deliver Larrabee-based discrete graphics products have been put on hold. News like this is hard to interpret. Is this a signal that Larrabee's importance will be marginal? Or should developers of HPC and non-graphics software see here a unique opportunity to enjoy undisputed attention from Intel's Larrabee team?

We developers don't enjoy much rummaging through the news. We are happier with final specifications, reliable benchmarks, and stable tools to play with. In this sea of dubious, fragmented and sometimes contradictory information, two events stood out. The first is Michael Abrash's article "A First Look at the Larrabee New Instructions (LRBni)" in *Dr. Dobb's* (http://www.ddj.com/architect/216402188), which finally detailed the new LRBni instructions. That article is, I bet, on its way to become the most cited non-peer-reviewed publication among academic, peer-reviewed papers of 2009–2010 in the high-performance computing field. The second is the release, last June, of Intel's C++ Larrabee Prototype Library (http://software.intel.com/en-us/articles/prototype-primitives-guide/), a simple header file that — all the no-warranty disclaimers factored in — provides the actual prototypes of the intrinsics associated to new instructions, along with reference semantics detailed in plain C code. This prototype library allows you to write, today, code that will run without major changes on the real hardware. The promise is that someday a compiler will read these intrinsics and emit corresponding LRBni instructions. This makes the library a convenient prototyping tool, especially if you have limited experience with writing SIMD code.

Unfortunately, the library has a major weakness: It won't give you the slightest hint on the performance of the code you write. If you work in high-performance computing, you might be familiar with profiling and simulating your code, estimating its latency and throughput, and measuring indicators like cache misses ratios and Clocks Per Instruction (CPI). This library won't allow you to do any of these.

This is, alas, a big question mark. With instructions of unprecedented complexity, which may potentially decompose into hundreds of microinstructions and entail up to 16 memory references, each taking even 800 clock cycles to complete in case of cache misses, estimating the performance of LRBni code on paper is a major challenge.

What's Under the Hood

After the onset of the multicore revolution, chip designers are using a higher number of simpler, smaller, lower-clocked, more energy efficient cores rather than pushing the limits of a larger, power hungry, highly speculative, and less efficient single core.

Larrabee will include 32 or more cores, each of which is much simpler than the Pentium 4, and significantly simpler than the current Intel Core2 or i7 architecture. Cores will process instructions in order, sending out-of-order execution to history as a silicon-hungry and power-inefficient luxury of the past. Not only will Larrabee save on all the circuitry required for out-of-order execution, rumor has it that branch prediction will be simpler and branch misprediction penalties might cost up to 9 cycles.

Each core features 512-bit wide SIMD units for Larrabee. Cores will have similar traits as in other multicore architectures, i.e., relatively small amounts of local memory (256 kbyte per core), a high number of registers (128), a ring interconnect. Even the latency scaling among memory levels will be similar to earlier many-cores.

When developing compute-intensive code on Larrabee, developers should keep in mind two fundamental principles:

- Transform control flow into data flow
- Use an explicit working set

The first principle is crucial for avoiding expensive misprediction penalties. More important, branch elimination is essential to perform SIMD parallelization. The second principle allows for the best exploitation of the (small) per-core local memory. An explicitly streaming design avoids thrashing the caches, and a careful use of non-temporal hints won't pollute the caches with single-use data.

Some less-than-pleasant manual unrolling of subsequent iterations of a loop (I call this "vertical unrolling") should still be able to pay off on Larrabee, given the relative abundance of registers. But at least, no horizontal unrolling should be needed: Horizontal unrolling consists of merging together loops that originally processed independent streams (see High-performance Regular Expression Scanning on the Cell/B.E. Processor [5, Section 5.6] at http://domino.research.ibm.com/comm/research_people.nsf/pages/scarpazza.pubs.html/$FILE/2009-06-ICS-scarpazza.pdf), in order to fill the read-after-write dependency stalls within each loop iteration that the compiler can't fill. The reason why no horizontal unrolling should be needed is the 4-way Simultaneous Multi-Threading (SMT) that Larrabee cores feature: The hardware avoids stalls by switching effortlessly to whichever other thread is ready to compute.

Additionally, Larrabee has masked scatter/gather instructions. Their impact is:

- Code economy: One gather instruction may replace hundreds of scalar instructions. To convince yourself, try expressing a gather's equivalent in your favorite SIMD assembly. Remember to unpack 16 indexes from the SIMD register, do pointer arithmetic, perform 16 loads, and repack the 16 read values into the SIMD result register. If the gather is masked, prepend each of the 16 loads with tests and jumps. If needed, include type conversion/promotion code. Good luck.
- No need for explicit packing and unpacking: As noted above, a gather instruction dereferences each 32-bit index (or pointer) in the *i*th lane of vector register and returns the result in the same *i*th lane of the result vector register. On other SIMD architectures, you must manually unpack each pointer to a scalar register or rotate it to a predefined location before use.
- Masking: No need for speculated writes. One technique to avoid expensive branches is to write output speculatively [5, Section 5.4]. This means using (unconditional, unpredicated) store instructions that always write something. When not needed, they write invalid output to ignored locations. You shouldn't ever have to do this on Larrabee because scatter instructions allow you to mask each of the 16 stores. The hardware skips the writes you don't want, and will likely complete earlier if only a few are selected.

The rest of this article is an in-depth walk-through of a single piece of code, ported to LRBni. The code I show is a simplified replacement for the kernels generated by flex, and you can use it to write your own high-performance lexical scanner.

The example I present here is, if you want, a lazy port to LRBni of that tokenizer. Given its intended use (search engine indexing), it is oriented to yield high throughput on large numbers of independent streams, rather than low execution time on a single stream.

Show Me Some Code Already!

Enough with the preliminaries, it's time to get our hands dirty. My example runs through the four pieces of code in Listings 1 through 4. These excerpts describe a simple tokenizer, to be used in a search engine indexer after case folding, not much different from the actual tokenizer that Lucene (http://lucene.apache.org/), the most popular open-source search engine library, uses internally. Listing 1 shows the flex specification; Listing 2 contains the tokenizer kernel automatically generated by flex; Listing 3 is a rewrite that I find simpler to read and optimize; and, finally, Listing 4 shows my parallel implementation that exploits SIMD LRBni intrinsics.

Listing 1 is my specification that tells flex what patterns I'm looking for, and what actions to perform when patterns match. Note the three sections, separated by "%%" lines (if you are unfamiliar with this syntax, take a quick look at flex's manual at http://flex.sourceforge.net/manual/). The first section defines patterns such as letters, digits, alphabetic and alphanumeric strings, e-mail addresses, company names, and stop words (i.e., words that you want to filter out). The second section says that we are looking for three classes of tokens: (1) stop words and stray single characters, which we ignore; (2) regular tokens that we keep unchanged; and (3) acronyms, which we emit with a special flag, so that they are marked for some post-processing (e.g. dot removal) in later stages. The third section, omitted here, would be the C implementation of utility functions like *emit_token*.

```
LISTING ONE
LETTER          [a-z]
DIGIT           [0-9]
P                ("_"|[,-/])
HAS_DIGIT      ({LETTER}|{DIGIT})*{DIGIT}({LETTER}|{DIGIT})*
ALPHA           {LETTER}+
ALPHANUM        ({LETTER}|{DIGIT})+
ACRONYM         {ALPHA}"."({ALPHA}".")+
COMPANY         {ALPHA}("&"|"@"){ALPHA}
EMAIL           {ALPHANUM}(("."|"-
"|"_"){ALPHANUM})*"@"{ALPHANUM}(("."|"-"){ALPHANUM})+
HOST             {ALPHANUM}("."{ALPHANUM})+
NUM
{ALPHANUM}{P}{HAS_DIGIT}|{HAS_DIGIT}{P}{ALPHANUM}|{ALPHANUM}({P}{
HAS_DIGIT}{P}{ALPHANUM})+|{HAS_DIGIT}({P}{ALPHANUM}{P}{HAS_DIGIT}
)+|{ALPHANUM}{P}{HAS_DIGIT}({P}{ALPHANUM}{P}{HAS_DIGIT})+|{HAS_DI
GIT}{P}{ALPHANUM}({P}{HAS_DIGIT}{P}{ALPHANUM})+
STOPWORD
"a"|"an"|"and"|"are"|"as"|"at"|"be"|"but"|"by"|"for"|"if"|"in"|"i
nto"|"is"|"it"|"no"|"not"|"of"|"on"|"or"|"s"|"such"|"t"|"that"|"t
he"|"their"|"then"|"there"|"these"|"they"|"this"|"to"|"was"|"will
"|"with"
KEPT_AS_IS     {ALPHANUM}|{COMPANY}|{EMAIL}|{HOST}|{NUM}
%%
{STOPWORD}|.|\n     /*do nothing*/;
{KEPT_AS_IS}         emit_token(CLASS_TOKEN, yytext);
{ACRONYM}            emit_token(CLASS_ACRONYM, yytext);
%%
/* C code omitted */
```

You might remember from your college days that a finite-state machine (FSM) is sufficient to recognize a regular language. Flex

generates that machine, in the form of C code, from the specification in Listing 1. That C code contains (among a sea of nitty-gritty details) two main blocks: The state transition table of the state machine (split in variables *yy_nxt* and *yy_accept*), and the C kernel that runs that machine, summarized in Listing 2. Don't be scared, I don't expect you to dive into this code, I just want to point out a few details. This code loops over the input, and it is divided into an inner loop and a switch block. The inner loop runs an FSM using the state transition table *yy_nxt* until it finds a valid pattern (i.e., the input characters between pointers *yy_bp* and *yy_cp*). When that happens, the switch block performs an appropriate action. Note that flex has preserved the actions we indicated in the specification, and they appear here as case 1, 2, and 3.

LISTING TWO

```
while ( 1 ) /* loops until end-of-file is reached */
  {
    yy_cp = (yy_c_buf_p);

    /* Support of yytext. */
    *yy_cp = (yy_hold_char);

    /* yy_bp points to the position in yy_ch_buf of the start of
     * the current run. */
    yy_bp = yy_cp;
    yy_current_state = (yy_start);
  yy_match:
    while ( (yy_current_state = yy_nxt[yy_current_state] [
YY_SC_TO_UI(*yy_cp) ]) > 0 )
      {
          if ( yy_accept[yy_current_state] )
            {
              (yy_last_accepting_state) = yy_current_state;
              (yy_last_accepting_cpos) = yy_cp;
            }

          ++yy_cp;
      }
      yy_current_state = -yy_current_state;

  yy_find_action:
    yy_act = yy_accept[yy_current_state];

    YY_DO_BEFORE_ACTION;

  do_action:      /* This label is used only to access EOF actions. */

    switch ( yy_act )
      { /* beginning of action switch */
      case 0: /* must back up */
          /* undo the effects of YY_DO_BEFORE_ACTION */
          *yy_cp = (yy_hold_char);
          yy_cp = (yy_last_accepting_cpos) + 1;
          yy_current_state = (yy_last_accepting_state);
          goto yy_find_action;

      case 1:
          /* rule 1 can match eol */
          YY_RULE_SETUP
          /*do nothing*/;
        YY_BREAK

      case 2:
          YY_RULE_SETUP
          emit_token(CLASS_TOKEN, yytext);
        YY_BREAK

      case 3:
          YY_RULE_SETUP
          emit_token(CLASS_ACRONYM, yytext);
        YY_BREAK
            ...
      }
  }
```

The purpose of case 0 is related to the concept of back-up, and it requires a little explanation. Consider that flex's tokenizers always try to find the longest match; for example, when looking for pattern "*i[a-z]*n*", the string "internationalization" matches, dominating all its shorter substrings ("in", "intern", "internation", "ionalization", "ization", etc.), which are ignored. This feature is called longest-of-the-leftmost semantics. To implement it, the tokenizer may delay acting on a valid pattern; rather, it saves the current state and position in the input (the *if* statement within the inner *while* loop), and continues examining more input in an attempt to find a longer match. Sometimes the attempt succeeds, sometimes not. When it fails, the machine falls into a "back-up state", and reaches case 0. There, the machine restores the last saved accepting position, and executes the saved associated action.

For my convenience I rewrite that kernel in the form of Listing 3. The new form is easier to read and uses a single table *yy_next*, where each entry (a state code) contains both a state number and condition flags. The machine behavior is now regulated by condition flags. Whenever a match is found, the machine enters a final state, i.e., BIT_FINAL is active in the state code. If a token is matched (as opposed to a stop word), then BIT_TOKEN is active. If the token is an acronym, BIT_ACRONYM is active (to support more token types, you just need to reserve more flag bits). Finally, the save/back-up mechanism is cleanly regulated by BIT_SAVE and BIT_RESTORE flags. I wrote a simple utility to convert tables *yy_nxt* and *yy_accept* from the format that flex generates to table *yy_next*, in the format I just described. Also, you might have noticed that Listing 3 only saves and restores the last accepting position, and not the associated action. This is the result of an optimizing manipulation of the state machine, which is beyond the scope of this article [5, Section 5.5].

LISTING THREE

```
void tokenize ( unsigned char * const input,
                    unsigned char * const input_end)
{
  state_t current_state = 0; // initial state

  unsigned char * yy_bp = input; // base pointer
  unsigned char * yy_cp = input; // current pointer

  unsigned char * last_accepting_cpos   = (unsigned char*)
UNKNOWN;

  while (1) {
    const unsigned char in_chr = *yy_cp;
    const state_t next_code  = yy_next[current_state * ALPHABET
+ in_chr];
    const state_t next_state = next_code >> 8 /* last 8 bits
reserved for flags */;

    const bool is_final    = next_code & BIT_FINAL;
    const bool is_save     = next_code & BIT_SAVE;
    const bool is_restore  = next_code & BIT_RESTORE;
    const bool is_token    = next_code & BIT_TOKEN;
    const bool is_acronym  = next_code & BIT_ACRONYM;

    if (! is_final ) {
      ++yy_cp;
      if ( is_save ) last_accepting_cpos   = yy_cp;
      if ( yy_cp >= input_end ) goto the_end;
    } else {
      if ( is_restore )
          yy_cp = last_accepting_cpos;

      if (is_token)
          emit_token(CLASS_TOKEN + is_acronym, yy_bp, yy_cp);

      yy_bp = yy_cp;
    }
    current_state = next_state;
  }

 the_end:
  ;
}
```

The code you see in Listing 3 is sequential (as opposed to parallel) and scalar (no SIMD). I like to say it operates on one lane. To exploit Larrabee's SIMD instructions, we extend this code to operate on the 16 lanes that Larrabee's 512-bit registers offer. The result is in Listing 4.

LISTING FOUR

```
#define LANES 16

typedef union {
  __m512i mm;
  __m512  mf;
  __m512d md;
  uint32_t u32[16];
} variant512_t;

#define EXTRACT(_x_,_i_) ({ variant512_t t = {mm:_x_};
t.u32[_i_];})
#define CAST_F2I(_x_)    ({ variant512_t t = {mf:_x_}; t.mm;})
#define CAST_I2F(_x_)    ({ variant512_t t = {mm:_x_}; t.mf;})

inline __m512i _mm512i_mask_movd(__m512i v1_old, __mmask k1,
__m512i v2)
{
  return CAST_F2I( _mm512_mask_movd( CAST_I2F(v1_old), k1,
CAST_I2F(v2)) );
}

typedef struct {
  uint32_t start;
  uint32_t stop;
  uint32_t type;
  uint32_t unused;
} tt_entry_t;

//unified output token table
tt_entry_t ttable_all  [ TEST_TOKEN_TABLE_SIZE * LANES ]
align64;

tt_entry_t * ttp_start [ LANES ]; // beginning of per-automaton
output table partition
tt_entry_t * ttp_end   [ LANES ]; // end of per-automaton output
table partition
tt_entry_t * ttp_used  [ LANES ]; // end of valid data in the
output table partition

void tokenize ( const unsigned char * const input_starts
[LANES],
                const unsigned char * const input_ends
[LANES])
{
  const unsigned input_base  = 0;          /* zero for the
moment, useful when the input will be above 4GB */
  void * const   stt_base    = yy_next;

  __m512i mm_current_states      = _mm512_set_1to16_pi( 0 );
  __m512i mm_next_states;

  //////// CONSTANTS
////////////////////////////////////////////////////////////
////////////////////////
  const __mmask mask_all_ones       = 0xFFFF; /* sixteen bits,
all ones, used to check collective termination  */
  const __m512i mm_zeroes            = _mm512_set_1to16_pi( 0 );
  const __m512i mm_ones              = _mm512_set_1to16_pi( 1 );
  const __m512i mm_twos              = _mm512_set_1to16_pi( 2 );
/* to increment tteptr */

  const __m512i mm_final_mask       = _mm512_set_1to16_pi(
BIT_FINAL   );
  const __m512i mm_save_mask        = _mm512_set_1to16_pi(
BIT_SAVE    );
  const __m512i mm_restore_mask     = _mm512_set_1to16_pi(
BIT_RESTORE );
  const __m512i mm_token_mask       = _mm512_set_1to16_pi(
BIT_TOKEN   );
  const __m512i mm_acronym_mask     = _mm512_set_1to16_pi(
BIT_ACRONYM );
  const __m512i mm_kill_flags_mask  = _mm512_set_1to16_pi(
0xFFFFFF00  );

  const __m512i mm_SR_ofs       = _mm512_set_1to16_pi( 8 /*
shift right 8 bits */ );
  assert(ALPHABET == 128 /* if it is not, change the 7 below
*/);
  const __m512i mm_SL_alpha     = _mm512_set_1to16_pi( 7 ); //
for shift left 7 ( == mul 128) operations

////////////////////////////////////////////////////////////
////////////////////////////////////////

  __m512i mm_bp    = * (const __m512i*) input_starts;
  __m512i mm_cp    = * (const __m512i*) input_starts;
  __m512i mm_lacp  = * (const __m512i*) input_starts;
  __m512i mm_eob   = * (const __m512i*) input_ends;

  /* quadword index into the token table used, for use with
scatterd */
#define TTEP_IDX(_dfa_) ( U32( (ttp_used[ (_dfa_)]-ttable_all) )
* sizeof(tt_entry_t) / (4 /*SCALE*/) )

  __m512i mm_tte_idx = { TTEP_IDX( 0), TTEP_IDX( 1), TTEP_IDX(
2), TTEP_IDX( 3),
                         TTEP_IDX( 4), TTEP_IDX( 5),
TTEP_IDX( 6), TTEP_IDX( 7),
                         TTEP_IDX( 8), TTEP_IDX( 9),
TTEP_IDX(10), TTEP_IDX(11),
                         TTEP_IDX(12), TTEP_IDX(13),
TTEP_IDX(14), TTEP_IDX(15) };

  while (1) {
    const __m512i mm_in_chars = CAST_F2I( _mm512_gatherd( mm_cp,
input_base, _MM_FULLUPC_UINT8I, _MM_SCALE_1, _MM_HINT_NONE ));

    const __m512i mm_states_xalpha      = _mm512_sll_pi (
mm_current_states, mm_SL_alpha );
    const __m512i mm_next_code_indexes = _mm512_add_pi (
mm_states_xalpha, mm_in_chars );
    const __mmask mask_not_eob          =
_mm512_cmpnle_pu(mm_eob, mm_cp); // "not at end-of-buffer"
condition;
    const __mmask mask_eob              = ~mask_not_eob;
// "at end-of-buffer"      condition

          __m512i mm_next_codes         = CAST_F2I (
_mm512_mask_gatherd( CAST_I2F(mm_next_codes), mask_not_eob,
mm_next_code_indexes,

                     stt_base, _MM_FULLUPC_NONE, _MM_SCALE_4,
_MM_HINT_NONE ));
    const __m512i mm_next_states        =  _mm512_sra_pi (
mm_next_codes, mm_SR_ofs);

    const __mmask mask_final            = _mm512_cmpnle_pu(
_mm512_and_pi ( mm_next_codes, mm_final_mask    ),  mm_zeroes);
    const __mmask mask_save             = _mm512_cmpnle_pu(
_mm512_and_pi ( mm_next_codes, mm_save_mask     ),  mm_zeroes);
    const __mmask mask_restore          = _mm512_cmpnle_pu(
_mm512_and_pi ( mm_next_codes, mm_restore_mask ),  mm_zeroes);
    const __mmask mask_token            = _mm512_cmpnle_pu(
_mm512_and_pi ( mm_next_codes, mm_token_mask    ),  mm_zeroes);

    const __m512i mm_are_acronym        = _mm512_and_pi (
mm_next_codes, mm_acronym_mask ); /* can be more than 1 bit */
    mm_next_codes = _mm512_and_pi ( mm_next_codes,
mm_kill_flags_mask );
    const __mmask mask_not_final        = ~ mask_final;

    mm_cp              = _mm512_mask_add_pi ( mm_cp,
mask_not_final & mask_not_eob,  mm_cp, mm_ones);
    mm_current_states = _mm512i_mask_movd  ( mm_current_states,
mask_not_eob, mm_next_states );
    mm_lacp            = _mm512i_mask_movd  ( mm_lacp,
mask_save,    mm_cp          );
    mm_cp              = _mm512i_mask_movd  ( mm_cp,
mask_restore, mm_lacp        );

    if ( mask_eob == mask_all_ones ) goto the_end; // all
automata are complete

    /* selectively commit results into the output token tables
*/
    _mm512_mask_scatterd(ttable_all, mask_token, mm_tte_idx,
CAST_I2F(mm_bp),           _MM_DOWNC_NONE, _MM_SCALE_4,
_MM_HINT_NT );
    mm_tte_idx = _mm512_mask_add_pi ( mm_tte_idx, mask_token,
mm_tte_idx, mm_ones );
    _mm512_mask_scatterd(ttable_all, mask_token, mm_tte_idx,
CAST_I2F(mm_cp),           _MM_DOWNC_NONE, _MM_SCALE_4,
_MM_HINT_NT );
    mm_tte_idx = _mm512_mask_add_pi ( mm_tte_idx, mask_token,
mm_tte_idx, mm_ones );
    _mm512_mask_scatterd(ttable_all, mask_token, mm_tte_idx,
CAST_I2F(mm_are_acronym), _MM_DOWNC_NONE, _MM_SCALE_4,
_MM_HINT_NT );
    mm_tte_idx = _mm512_mask_add_pi ( mm_tte_idx, mask_token,
mm_tte_idx, mm_twos /*1 + padding*/);

    mm_bp = _mm512i_mask_movd( mm_bp, mask_final, mm_cp);
  }
```

```
 the_end: ;
  // export token table pointers to global vars
  const unsigned quadwords_per_entry = sizeof (tt_entry_t) /
sizeof (uint32_t);
  for ( unsigned dfa ; dfa<LANES ; dfa++) {
    const unsigned n_entries = EXTRACT(mm_tte_idx, dfa) /
quadwords_per_entry;
    ttp_used[ dfa] = ttable_all + n_entries;
  }
}
```

In the preamble, I define a *variant512_t* union to allow convenient access to the contents of a 512-bit variable, as if it were a SIMD vector of integers, floats, or doubles, or as a C array of 32-bit unsigned integers. Thanks to this union, I can write macros (lines 10–12) to extract elements and cast float vectors to integer vectors and vice versa, using clean, pointer-less GCC code (beware of my use of statement expressions; see http://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/Statement-Exprs.html). Other methods use brute-force pointer casting and dereferencing, and have disadvantages: They hijack the language type safety and mess with the compiler's type-based pointer aliasing rules (with noisy warnings and potentially incorrect code).

The need to cast SIMD vectors and the need for a masked move that operates on integer SIMD vectors (my *_mm512i_mask_movd* function) arise from Intel's choice to define gather and scatter intrinsics only for float vectors. I use a CAST_I2F before a scatter, to pretend I'm storing floating points, and a CAST_F2I after a gather. It's okay, Intel, we non-numerical programmers are used to being treated as a "secondary audience". We have been used to this since the days of MMX and SSEx.

The code explosion from Listing 3 to Listing 4 is common when doing SIMDization. The code grows for two reasons. First, we write at a lower level, where C operators like +, & and > become verbose intrinsics like *_mm512_add_pi, _mm512_and_pi,* and *_mm512_cmpnle_pu*. Except for manual register allocation, this is pretty much like writing assembly code. We have also inlined function *emit_token*.

A second, more profound, reason is that we have replaced control flow with data flow. With the exception of the *if* that verifies the termination condition, the 16 FSMs on the 16 lanes are processed by the same instructions, and therefore must follow the control flow. They can't diverge like separate threads, they can't take separate paths, they can't have their own *if*s, *while*s, and *switch*s. Each iteration in the *while* loop describes 16 FSMs, all simultaneously consuming input, performing a state transition and possibly generating output. The execution of these FSMs is conjoined like the life of 16 siamese twins (called "sexdecuplets", if Wikipedia can be trusted). This is true to the extent that if one FSM gets to the end of its input, it must keep iterating until all the other FSMs also complete.

Each FSM sets a bit in variable *mask_eob* upon its end of buffer; when the mask is all ones, we are done. This is not a major issue and can be solved without performance degradation, but not without making the code more complex and difficult to explain.

As a consequence, we must rewrite the FSM's choices as branchless expressions. We use a technique sometimes called software-level speculation: We compute the result of both sides of a branch and then choose the relevant one with a selection instruction (and this takes more lines of code than the original branch.) On the Cell, you use the *spu_sel* intrinsic; on Larrabee, you use masked instructions. To see this technique in action, note the three masked scatter instructions in lines 109, 111, and 113. For each FSM, they store the values of *bp, cp,* and *token_type* at the end of that FSM's own token table. But this only happens if that FSM has matched a valid token in this iteration, as reflected by the respective bit in *mask_token*. Three associated masked adds (lines 110, 112, 114) conditionally advance the end-of-table indexes (*mm_tte_idx*), for the only automata that wrote output. Similarly, we turn the updates of *bp, cp,* and *last_accepting_cpos* (that appeared under *if* conditions in Listing 3) into masked adds or masked moves (lines 101–104).

Except for the *if* statement that detects the global termination condition, the code is branchless and it corresponds to the data flow of Figure 1. The figure represents precisely one iteration of the loop in Listing 4, where 16 conjoined FSMs read the respective inputs, load the next state codes from the state transition table, decode them (possibly generating outputs) and jump to the next states. A black box represents a vector variable, a grey box represents a constant, a red box represents a mask, and a blue circle represents a vector operation.

You can unroll this code *ad libitum* with no special prologues or epilogues, and no impact on correctness. To increase performance, you can also remove the *if* of the termination condition everywhere but in the last unrolled iteration.

Conclusion

I have shown how to port a simple, sequential piece of code into a fully parallel, SIMDized version that uses Larrabee's new instructions. The effort required for this data-parallel redesign is not trivial, but I don't believe that it is beyond the reach of programmers already involved in crafting hand-optimized code.

This kind of human effort is crucial because compilers won't likely be able to deliver a similar quality of result. Furthermore, it is more important on Larrabee than on previous Intel machines because Larrabee's SIMD width is four times larger than the usual 128-bit SIMD units: As a consequence, non-SIMD code will likely leave a much higher performance fraction on the table.

On the bright side, coding with LRBni intrinsics seems more natural and less verbose than coding with SSE intrinsics. One iteration of the loop in the code I presented needs approximately 35 instructions to transition 16 finite-state machines (approximately 2.2 instructions/transition), while other 128-bit SIMD instruction sets require at least 100 instructions to transition 4 machines (approximately 25 instructions/transition).

Don't get me wrong. Without a clue on instruction latencies, you cannot translate these instruction economy statistics into performance figures. At this time, nobody except Intel may estimate the amount of clock cycles taken by any LRBni instruction. Scatter/gather instructions, for example, will likely be decomposed

in multiple pointer arithmetic and store/load microinstructions, which might take a high cumulative number of clock cycles to complete. The performance of this code depends on how successful Intel engineers will be in squeezing LRBni instructions into a handful of clock cycles. It's not an easy task, especially for complex instructions like scaled, masked scatter/gather with type conversion.

Acknowledgments

Thanks to Sally A. McKee, Jamin Naghmouchi, Michael Perrone and Greg Pfister for their useful comments.

Disclaimer: Any claim or information reported here on Larrabee or other Intel products may not be final or reliable. The author assumes no liability for the use or interpretation of information contained herein. This article reflects the views and the

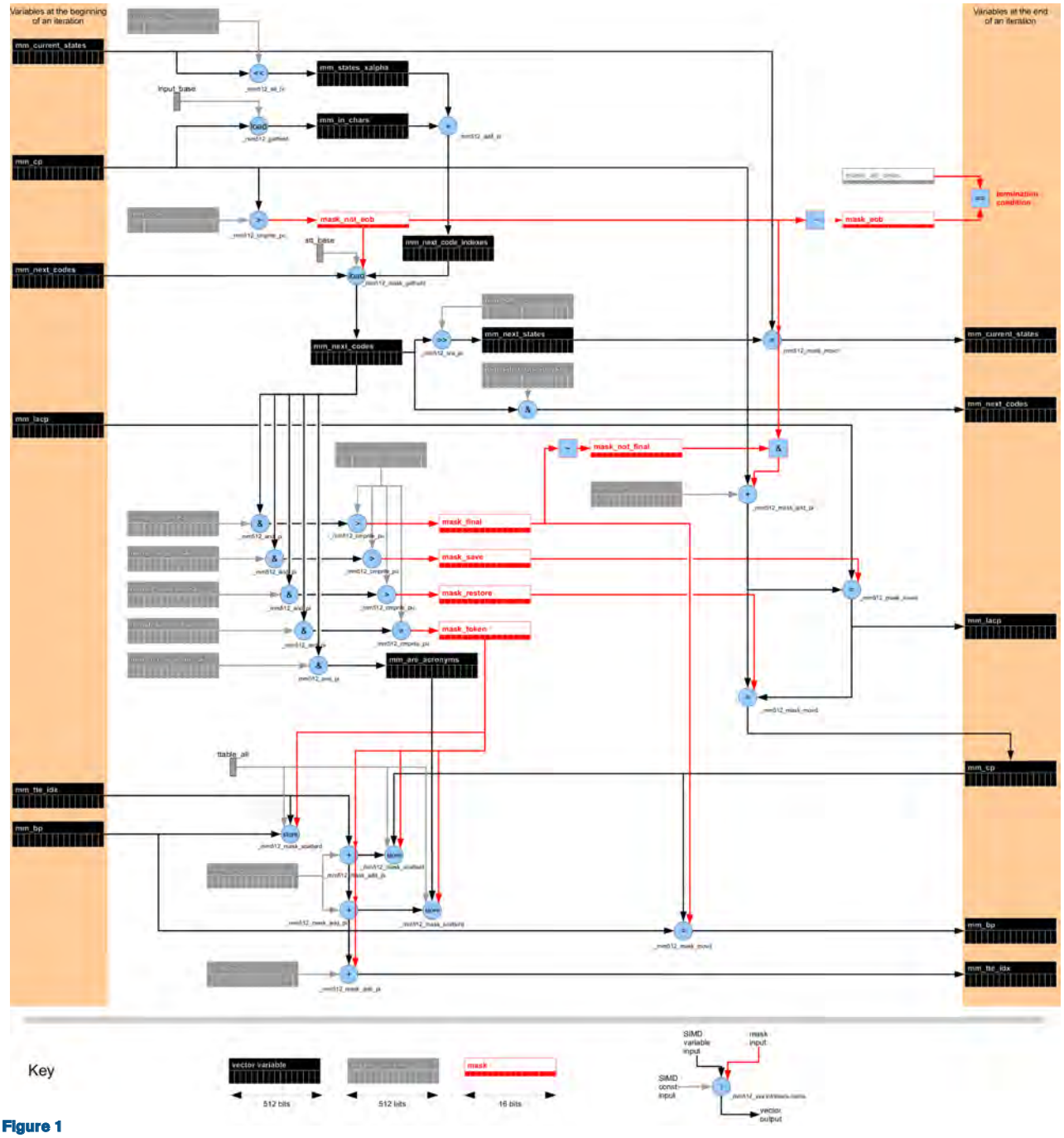

Figure 1

opinions solely of the author, which may not necessarily be endorsed or approved by IBM.

References

[1] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, 2008. "Larrabee: A Many-Core x86 Architecture for Visual Computing", ACM Transactions on Graphics, 27, 3, 2008. http://portal.acm.org/citation.cfm?doid=1360612.1360617

[2] M. Abrash, "A First Look at the Larrabee New Instructions (LRBni)", *Dr. Dobb's*, April 1st, 2009. http://www.ddj.com/hpc-high-performance-computing/216402188

[3] Intel Software Network, "C++ Larrabee Prototype Library", June 19, 2009. http://software.intel.com/en-us/articles/prototype-primitives-guide/

[4] K. Asanovic, R. Bodik, J. Demmel, J. Kubiatowicz, K. Keutzer, E. Lee, G. Necula, D. Patterson, K. Sen, J. Shalf, J. Wawrzynek, K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley". http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/David%20Patterson.pdf

[5] D. P. Scarpazza, G. F. Russell, "High-performance Regular Expression Scanning on the Cell/B.E. Processor", 23rd International Conference on Supercomputing (ICS'09), IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, June 2009. http://domino.research.ibm.com/comm/research_people.nsf/pages/scarpazza.pubs.html/$FILE/2009-06-ICS-scarpazza.pdf

[6] D. P. Scarpazza, G. W. Braudaway, "Workload Characterization and Optimization of High-performance Text Indexing on the Cell Processor", IEEE International Symposium on Workload Characterization (IISWC'09), Austin, TX, October 4, 2009. http://domino.research.ibm.com/comm/research_people.nsf/pages/scarpazza.pubs.html/$FILE/2009-10-04-IISWC-scarpazza.pdf

[7] The Apache Software Foundation. Lucene. http://lucene.apache.org

[8] V. Paxson, flex: a fast lexical analyzer generator. http://flex.sourceforge.net/manual/

[9] The Free Software Foundation, Using the GNU compiler collection (GCC), "Section 5.1: Statements and Declarations in Expressions". http://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/Statement-Exprs.html

— Daniele Paolo Scarpazza is a member of the Multicore Computing Department at the IBM T. J. Watson Research Center. He is the author of articles and scholarly papers on parallel algorithms and performance optimization on emerging multicore architectures.

Return to Table of Contents

Cloud Computing: Detecting Scalability Problems

Intel's Parallel Universe Portal is a free developer's service in the cloud

by Gastón Hillar

You already know that achieving a linear speedup as the number of cores increases in real-life parallelized applications is indeed very difficult. However, sometimes the multicore scalability of certain algorithms for existing multicore systems could be worse than expected. The overhead and the bugs introduced by concurrency could bring really unexpected scalability problems when the number of cores increases. Intel can help you with its Parallel Universe Portal, a free service in the cloud.

You don't have access to a system with an Intel Xeon X5560 CPU (http://www.intel.com/p/en_US/products/server/processor/xeon5000) running at a maximum clock speed of 2.80 GHz and offering 8 Hyperthreaded physical cores (16 logical cores, 16 hardware threads). However, you want to test the multicore scalability for an algorithm as the number of cores increases, up to 16 logical cores. If your application is written in C++, you can compile it in 32-bits to run on Windows, and the total time needed to run it using 1, 2, 4, 8, and 16 logical cores is less than 1 minute, you can use the new scalability service offered by Intel Parallel Universe Portal (see http://www.ddj.com/windows/221800426). If you already own a system with 16 logical cores, you can skip this post.

It is very easy to use this new service. You just have to follow a few steps and you'll be able to check the multicore scalability of your application up to 16 logical cores. I'm going to explain you how to use this service step-by-step.

Create an application without the need of user interaction. Make the necessary changes to make sure your application will try to use as many logical cores as available in the underlying hardware. This way, you'll be able to test its multicore scalability. Once you obtain the first results, you will be able to make the necessary changes to test other specific scenarios.

If the application needs additional files or DLLs, copy all the necessary files to a new folder and create a ZIP file including the EXE file and all these additional files. For example, if the application is myparallelapp.exe and it needs the libiomp5md.dll DLL, your ZIP file should include both files. So far, the service doesn't support .NET applications, they must be unmanaged Windows C++ apps.

Enter Intel Parallel Universe Portal (http://paralleluniverse.intel.com/), login with your account, and click on Start Here.

Enter a name to identify your session. Click on the Upload button, choose the previously created ZIP file and the file will begin uploading, as in Figure 1.

Click on the Next button. It is time to configure your job. You have to select the command line to run, the executable file, and the arguments, as in Figure 2.

You can leave the arguments textbox blank if the application doesn't need arguments. Click on the Next button. The website will offer a summary with the session name, the uploaded zip file name, and the command line to run. You have to check whether everything is specified as expected, as in Figure 3.

If everything is okay, you have to click on the Submit button. This way, the job will enter in a

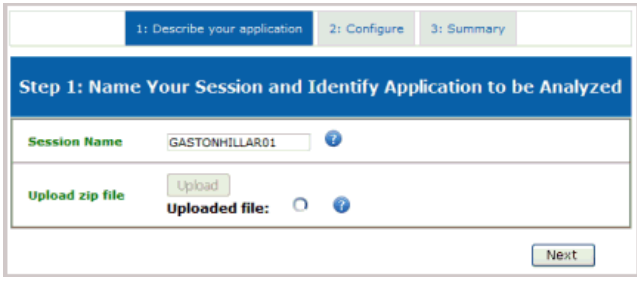

Figure 1: Uploading the ZIP file with the application to run and all the necessary additional files.

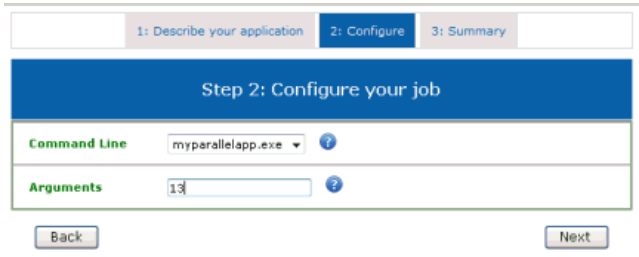

Figure 2: Configuring the job to run.

queue. The web page will show your job and it will refresh the information about it. However, you can also wait for an e-mail with information about your job. If something goes wrong, you'll receive an e-mail with a link to a web page with more information about the errors. The most common problem is that the application can take more than 1 minute to run with all the different configurations. As aforementioned, there's a 1 minute time limit. The other usual problem is an unsatisfied dependency, like a missing DLL that the application needs to run. Remember to include everything your app needs in the ZIP file. The error report usually includes the name of the unsatisfied dependencies.

If the application could run without problems using 1, 2, 4, 8, and 16 logical cores in less than 1 minute, you'll receive an e-mail with a link to a web page with an easy-to-understand scalability report. The report offers both tables and graphs with the following information for the different number of logical cores used:

- The elapsed time in seconds.
- The average concurrency.

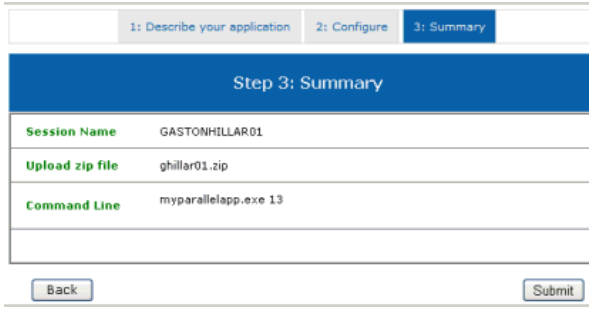

Figure 3: Checking the job's summary.

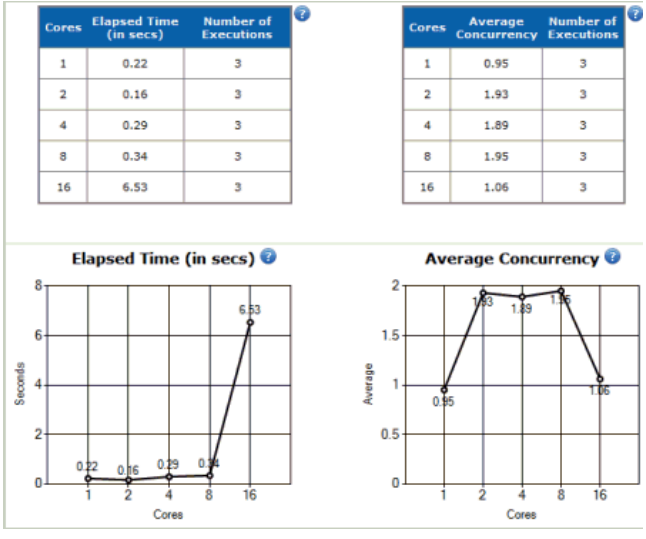

Figure 4: Tables and graphs with the elapsed time in seconds and the average concurrency for an application with multicore scalability problems.

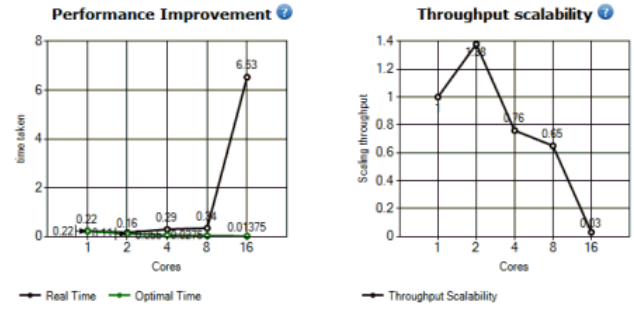

Figure 5: Graphs with the performance improvement and the throughput scalability for an application with multicore scalability problems.

Figure 4 shows part of the report created by a job with easy-to-detect multicore scalability problems. As you can see, the application takes less time to run when it moves from 1 to 2 logical cores. However, when it runs with 4, 8, and 16 logical cores, there's no speedup. In fact, when it tries to take advantage of 16 logical cores, its performance is really bad.

The application has a very important multicore scalability problem. It cannot achieve significant speedups when the number of cores is greater than 2. Besides, it introduces a very big overhead when the number of cores is 16. There is a very important overhead to work with 16 logical cores and the parallelized algorithms are taking more time than the time needed to complete the mission with just one logical core. Believe me; this situation is more common than expected. Redesigning serial algorithms in order to take advantage of multicore is a complex task. Redesigning them in order to scale as the number of cores is greater than 4 is even a more complex task. The manycore challenge is round the corner.

Besides, the report offers two additional graphs with the following information for the different number of logical cores used:

- The performance improvement, considering the time taken to complete the job with the different number of logical cores.
- The throughput scalability, explaining how the throughput scales as the number of logical cores increases.

Figure 5 shows the two graphs generated by the same job with multicore scalability problems. As you can see, the application increases both performance and throughput when it moves from 1 to 2 logical cores. However, when it runs with 4, 8, and 16 logical cores, throughput scalability decreases. The performance improvement graph shows two lines, a black line with the actual time needed to run the job with the different number of logical cores and a red line with the optimal time. In this case, the actual time needed to run the application with 16 logical cores is around 600 times the optimal time.

As you can see, the reports are very easy to understand. This application is not able to scale beyond 2 logical cores.

For more information, you can read also check Intel Parallel Universe Portal Forums at http://software.intel.com/en-us/forums/intel-parallel-universe-portal/.

You can also read "Beware of Those That Claim Linear Performance Increases" by Markus Levy at http://www.ddj.com/go-parallel/blog/archives/2009/06/beware_of_those.html.

If you want to use virtualization to test multicore scalability, you can read my article, "Using VirtualBox 3.0 Virtualization Software to Measure Multicore Scalability" at http://www.ddj.com/go-parallel/blog/archives/2009/07/using_virtualbo.html.

— Gastón Hillar is the author of C# 2008 and 2005 Threaded Programming: Beginner's Guide.

Return to Table of Contents

recls 100% .NET

Implementing a 100% C# implementation of recls for .NET.

by Matthew Wilson

Several years ago I wrote the column Positive Integration for *C/C++ Users Journal* and later *Dr. Dobb's Journal* (http://www.ddj.com/cpp/184401727?pgno=3), which discussed issues involved in adapting C/C++ libraries to other languages. The main exemplar project used was "recursive ls" or recls (http://synesis.com.au/software/recls/), a platform-independent recursive filesystem search library written in C and C++, and with a C API. Adaptation to numerous languages (including Ch, C#/.NET (via P/Invoke), D, Java, Python, and Ruby) was examined, covering the development of the library from versions 1.0 through 1.6. Since that time, the library has continued to evolve, and now stands at 1.8. A new C/C++ version, 1.9, will be released in the coming weeks.

I have long planned to rework the library implementation. The two main changes will be a substantial refactoring of the source files and packaging for the core library and the C++ layer, and a rewrite of some/all of the language mappings in the form of full "100%" implementations. This article describes the first of these, a 100% C# implementation of recls for .NET. For clarity, I'll refer to the original stream of work as recls 1.x and the new .NET library as recls 100% .NET in this article.

The reasons for these changes are:

- The core library has grown to a level of complexity such that I no longer find it easy to make changes
- I wanted to introduce diagnostic logging to the core library; this is included in recls 1.9
- I wanted to ease the burden of deployment. For example, with the .NET mapping in versions up to 1.8, the recls.dll exporting the core C API (for access via *P/Invoke*) must be manually packaged along with the C# API in recls.NET.dll. Automated tools (such as Visual Studio) do not automatically copy it to working areas. And organizational security policies may prohibit use of assemblies that call into "unmanaged" code.
- I wanted to take advantage of new features of languages over the last five years. As we'll see shortly, aspects of C# 3 make for improved syntax in client code for non-trivial search use cases
- I wanted to implement two long asked-for features: breadth-first search, and search-depth limiting. recls 1.x provides only depth-first search, and always does a full-depth search.

Despite being written entirely in C#, the implementation of recls 100% .NET is larger than can be fully covered here. So I intend to focus on the interesting design points, language features, and the differences in functionality between recls 1.x and recls 100% .NET.

API Differences

The first difference is a cosmetic one. To placate FxCop (http://en.wikipedia.org/wiki/FxCop), and also to clearly distinguish the new recls .NET API from the old for anyone who wishes to port their code to it, I changed the old recls namespace to Recls.

Similarly, the RECLS_FLAG enumeration is now *SearchOptions* (see Listing 1), and its enumerators are Files not FILES, Directories not DIRECTORIES, and so on. There are also fewer enumerators. Notably absent from the original are RECURSIVE, LINKS, DEVICES, NO_FOLLOW_LINKS, DIRECTORY_PARTS, DETAILS_LATER, PASSIVE_FTP, and ALLOW_REPARSE_DIRS. The changes reflect the intended increase in portability and improvements to discoverability and transparency of the new API, based on user feedback.

Listing 1: The SearchOptions enumeration

```
 [Flags]
public enum SearchOptions
{
  None                          = 0x00000000,
  Files                         = 0x00000001,
  Directories                   = 0x00000002,
  IgnoreInaccessibleNodes       = 0x00100000,
  MarkDirectories               = 0x00200000,
  IncludeHidden                 = 0x00000100,
  IncludeSystem                 = 0x00000200,
  DoNotTranslatePathSeparators = 0x00002000,
}
```

The *FileEntry* class is gone, replaced by the *IEntry* interface (see Listing 2). The *FtpSearch* class goes entirely, as the first version of recls 100% .NET does not support FTP search. The *DirectoryParts* class is no longer externally visible; the *DirectoryParts* getter-property now returns (an instance implementing) the interface *IDirectoryParts*; see Listing 3. The *FileSearch* class goes, and search is now provided by the (static) *FileSearcher* class.

Listing 2: The IEntry interface

```
// in namespace Recls
public interface IEntry
{
  string Path { get; }
  string SearchRelativePath { get; }
  string Drive { get; }
  string DirectoryPath { get; }
  string Directory { get; }
  string SearchDirectory { get; }
  string UncDrive { get; }
  string File { get; }
  string FileName { get; }
  string FileExtension { get; }
  DateTime CreationTime { get; }
  DateTime ModificationTime { get; }
  DateTime LastAccessTime { get; }
  DateTime LastStatusChangeTime { get; }
  long Size { get; }
  FileAttributes Attributes { get; }
  bool IsReadOnly { get; }
  bool IsDirectory { get; }
  bool IsUnc { get; }
  IDirectoryParts DirectoryParts { get; }
}
```

Listing 3: The IDirectoryParts interface

```
public interface IDirectoryParts
  : IEnumerable<string>
{
  int Count { get; }
  string this[int index] { get; }
  bool Contains(string item);
  void CopyTo(string[] array, int index);
}
```

IEntry vs. FileEntry

Table 1 compares the public interfaces of the old *FileEntry* class and recls 100% .NET's *IEntry* interface. The differences, highlighted in bold, involve changes to both syntax and semantics, and result from lessons learned by users of recls 1.x.

Drive changed from a character to a string so that there'd be less hassle when manipulating UNC-based paths: Now users can deal with a single property, rather than a drive letter character in one, and a (UNC) drive string in another. The spellings of *UNCDrive* and *IsUNC* changed to follow .NET idiom. The *Size* property changed from *ulong* to *long* to be CLS compatible (for example, to be able to be used from VB.NET and other .NET languages that don't support unsigned integral types). *IsLink* and *ShortFile* had to go by the wayside because of the need to be implemented 100% in terms of the CLR facilities (and not go to *P/Invoke*). The *Attributes* property was added to allow recls to stay relevant in light of evolution in the CLR of the file attributes that may be made available to managed programmers.

	old	**recls 100% .NET**
Methods	Dispose()	
	ToString() [returns Path]	ToString() [returns Path]
Properties	Path : string	Path : string
	SearchRelativePath : string	SearchRelativePath : string
	Drive : **char**	Drive : **string**
	Directory : string	Directory : string
	DirectoryPath : string	DirectoryPath : string
	SearchDirectory : string	SearchDirectory : string
	UNCDrive : string	**UncDrive** : string
	File : string	File : string
	ShortFile : string	
	FileName : string	FileName : string
	FileExt : string	
		FileExtension : string
	CreationTime : DateTime	CreationTime : DateTime
	ModificationTime : DateTime	ModificationTime : DateTime
	LastAccessTime : DateTime	LastAccessTime : DateTime
	LastStatusChangeTime : DateTime	LastStatusChangeTime : DateTime
	Size : **ulong**	Size : **long**
	IsReadOnly : bool	IsReadOnly : bool
	IsDirectory : bool	IsDirectory : bool
	IsLink : bool	
	IsUNC : bool	**IsUnc** : bool
	DirectoryParts : **DirectoryParts**	DirectoryParts : **IDirectoryParts**
		Attributes : FileAttributes

Table 1: Mappings Between Old and New Entry class/Interface Methods and Properties

There are also some semantic changes. The form of the file extension has changed, and now includes the dot, so "abc.net" will have an extension of ".net", rather than "net" as was the case with recls 1.x. Since this is a breaking change, I've removed the previous name, *FileExt*, and given it a new name *FileExtension*. (This also fits better with the .NET way of doing things, which is to avoid unnecessary contractions in names.)

It's useful to be able to paste the extension to another file name without having to pollute client code with logic to determine whether or not to insert the dot. Now, all of the following combinations will reproduce the full path (and, to be useful, may be used in combination with other strings to build correctly-formed new paths):

- DirectoryPath + File
- DirectoryPath + FileName + FileExtension
- Drive + Directory + File
- Drive + Directory + FileName + FileExtension

Specifying Search Criteria

The change from instance to class methods has probably the greatest impact on usage, so let's look at a pair of examples illustrating the old and the new. To search for all the font files in the Windows directory and its subdirectories with the old recls 1.x .NET mapping we'd write:

```
using recls;
foreach(FileEntry entry in new FileSearch(@"C:\Windows",
"*.fon|*.ttf", RECLS_FLAG.RECURSIVE))
{
  Console.WriteLine(entry);
}
```

With recls 100% .NET, you would write:

```
using Recls;
foreach(IEntry in FileSearcher.Search(@"C:\Windows",
"*.fon|*.ttf"))
{
  Console.WriteLine(entry);
}
```

With such a simple example the differences are not huge, which is a good thing. Nonetheless, two of the most significant changes are illustrated:

- Recursive search is now the default, so the RECLS_FLAG.RECURSIVE flag is not needed (or available)
- Search sequences are now obtained via static methods on a static class

A search is conducted from a specified directory, in which all entries (files or directories) that match the given pattern(s) and correspond to the given search options, up to a given depth, are retrieved. A directory may be absolute or relative, but must exist. If null (or the empty string) is specified, the current directory is assumed. A pattern may be a file (or directory) name, or may use wildcards, as in "*.ttf". Furthermore, the library supports multi-part patterns, allowing discovery of entries matching different wildcards within the same string, as in "*.fon|*.ttf". A null argument for the patterns parameter is interpreted to mean the "everything" pattern for the given platform (i.e. "*" on UNIX, and "*.*" on Windows).

The search options can select *Files*, *Directories*, or both; absence of both is interpreted as *Files*). Other options allow for tailoring the search policy, as follows:

- Searching of Hidden and/or System files, both of which are not normally listed
- Ignoring access-denied conditions, which would otherwise cause the search to be terminated, by specifying *IgnoreInaccessibleNodes*; this option is ignored if an exception handler is specified. This is often necessary when specifying *Hidden* and/or *System*.
- Marking of the *Path*, *SearchRelativePath* and *File* properties of entries that are directories (via *MarkDirectories*) with a trailing slash
- Preventing the automatic translation of per-platform pattern separators — ':' for UNIX; ':' for Windows — into the platform-independent pattern separator '|' (via *DoNotTranslatePathSeparators*)

Exceptions that interrupt the processing may be filtered by specifying an exception handler (see Listing 4).

Listing 4: Exception Handler Interface

```
enum ExceptionHandlerResult
{
  PropagateException = 0,
  ConsumeExceptionAndContinue
}
interface IExceptionHandler
{
  ExceptionHandlerResult OnException(string path, Exception x);
}
```

Returning *PropagateException* causes the exception to be rethrown, causing the search to be cancelled and the caller to receive the exception. Returning *ConsumeExceptionAndContinue* consumes the exception (perhaps after logging the condition) and continues the search, skipping the offending directory. Naturally, the purpose of this callback is not to allow users to attempt to suppress unrecoverable conditions, and the library does not invoke the callback in some such cases. Unfortunately, because the .NET exception hierarchy is such an abject mess, discriminating between logical errors, practically unrecoverable conditions, and recoverable runtime conditions is not a simple task, and it is likely that the set of exceptions made suppressible in this regard will change in future implementations. Users are expected to consume only specific expected exceptions; for instance, *System.IO.DirectoryNotFoundException*, rather than doing anything as unwise as consuming *System.Exception*.

Finally, processing a large directory tree with highly specific pattern(s) can lead to a user experience with discernible pauses, due to filesystem latencies. Consequently, a progress callback mechanism is also provided, in the form of the *IProgressHandler* interface (see Listing 5), which allows callers to be notified as each new (sub)directory is searched, perhaps to log the directory traversal changes to console, status bar, etc. It also affords the opportunity to apply search policy on a location basis, via return of a control code from the *ExceptionHandlerResult* enumeration: *CancelDirectory* causes the given directory and all its subdirectories to be excluded from the search; *CancelSearch* causes exclusion of all remaining directories, even those at a higher level in the tree.

Listing 5: Progress Handler Interface

```
enum ProgressHandlerResult
{
  Continue = 0,
  CancelDirectory,
  CancelSearch
}
interface IProgressHandler
{
  ProgressHandlerResult OnProgress(string directory, int depth);
}
```

Given this richness in search specification — directory, patterns, depth, options, exception-handler, progress-handler — there is clearly a conflict between flexibility and discoverability in the possible overloads for the file search functions. Ignoring parameter ordering and ambiguities caused by some parameters (directory and patterns) sharing the same type, there are 64 possible combinations of the six parameters. If we add in parameter ordering, it becomes 121. To get a handle on the problem consider the case for just three parameters, directory (*dir*), patterns (*ptns*) and depth, with and without parameter ordering considerations. (In both lists, type ambiguities are marked with a <-X->.)

Without considering parameter ordering, we have eight combinations, of which six are viable:

```
()
(dir)
(ptns) <-X-> (dir)
(depth)
(dir, ptns)
(dir, depth)
(ptns, depth) <-X-> (dir, depth)
(dir, ptns, depth)
```

If we add in parameter ordering, we get 16, of which nine are viable:

```
()
(dir)
(ptns) <-X->(dir)
(depth)
(dir, ptns)
(ptns, dir) <-X->(dir, ptns)
(dir, depth)
(ptns, depth) <-X->(dir, depth)
(depth, dir)
(depth, ptns) <-X->(depth, dir)
(dir, ptns, depth)
(ptns, dir, depth) <-X->(dir, ptns, depth)
(dir, depth, ptns)
(ptns, depth, dir) <-X->(dir, depth, ptns)
(depth, dir, ptns)
(depth, ptns, dir) <-X->(depth, dir, ptns)
```

You can imagine the complexity when permuting all six parameters! Clearly we need to make some judicious cuts. Since none of the parameters obviate the need for any of the others, the obvious must-have overload is one in which all six are present. The order is somewhat moot, but I'd suggest it should be either of the following:

```
(dir, ptns, options, depth, progressHandler, exceptionHandler)
(dir, ptns, options, progressHandler, exceptionHandler, depth)
```

Let's leave that decision for the moment while we consider the other options.

Another obvious decision is that we can throw out all permutations that don't have both directory and patterns parameters (and in that order). The utility of being able to specify only directory (to search for all files) or only patterns (to search in the current directory), rather simply specifying null in the stead of the omitted argument is vanishingly small. Not to mention the detraction from discoverability. So we can treat them as a mandatory unit. Furthermore, I think we can also stipulate that they'll always come first in the parameter list.

A further simplification that I felt was justified was that, as "advanced" options, we could treat the two handler arguments as a pair. The cost is a slight extra effort in specifying null for whichever is not needed, at the gain of reducing the overload set.

Given that it can make sense to specify depth independently of options, and them both independently of progress+exception, we can now cut the list down to eight (which ignores parameter ordering for the moment):

```
(dir, ptns)
(dir, ptns, options)
(dir, ptns, depth)
(dir, ptns, progressHandler, exceptionHandler)
(dir, ptns, options, depth)
(dir, ptns, options, progressHandler, exceptionHandler)
(dir, ptns, depth, progressHandler, exceptionHandler)
(dir, ptns, options, depth, progressHandler, exceptionHandler)
```

Another concern is that, so far, we've discussed the handlers in terms of the interfaces *IExceptionHandler* and *IProgressHandler*. But C# allows a different callback construct, the delegate, which is particularly useful with the advent of C# 2 and (even more so) with C# 3. recls 100% .NET defines two delegates, for handling exceptions and progress (see Listing 6).

Listing 6: Handler Delegates

```
public delegate ExceptionHandlerResult OnException(string path,
    Exception x);
public delegate ProgressHandlerResult OnProgress(string
    directory, int depth);
```

So, even if we thought eight was a survivable number of overloads (which is in doubt), providing for the delegate forms (which are highly convenient, as we'll see later on) would push this out to a minimum of 12. Unequivocally, this is too much choice, one of the enemies of discoverability.

Consequently, some hard decisions had to be made, and I made the necessary (and somewhat arbitrary) decisions to give the following overload set (where *{D}* designates a delegate form, as opposed to an interface form):

```
(dir, ptns)
(dir, ptns, options)
(dir, ptns, depth)
(dir, ptns, options, depth)
(dir, ptns, options, depth, progressHandler, exceptionHandler)
(dir, ptns, options, depth, progressHandler{D},
exceptionHandler{D})
```

Although six may still feel like a lot, the fact that C# discriminates between int and enumeration types makes it pretty easy to live with it without ambiguity. We couldn't do the same in C++.

There's one final refinement to the overloading. Even though a search involving progress and/or exception handler may not usually require depth, I chose to keep the parameter ordering consistent (i.e. depth follows options) as this is an established principle of interface design. It also fits in better with Visual Studio's Intellisense: When scrolling through the list of options, the additional parameters appear naturally at the end of the list, rather than jumping around confusingly. Because of these reasons, I decided to remove the third overload — (directory, patterns, depth) — giving a final five. You may demur, but I think these five overloads represent an appropriate balance between flexibility and discoverability.

The class interface for *FileSearcher* is shown in Listing 7.

Listing 7: FileSearcher class interface

```
public static class FileSearcher { // Properties public static
public static class FileSearcher
{
// Properties
  public static int UnrestricedDepth { get; }
  public static string WildcardsAll { get; }

// Search Operations
  public static IEnumerable<IEntry> Search(
    string directory
  , string patterns
  );
  public static IEnumerable<IEntry> Search(
    string directory
  , string patterns
  , SearchOptions options
  );
  public static IEnumerable<IEntry> Search(
    string directory
  , string patterns
  , SearchOptions options
  , int depth
  );
  public static IEnumerable<IEntry> Search(
    string directory
  , string patterns
  , SearchOptions options
  , int depth
  , IProgressHandler progressHandler
  , IExceptionHandler exceptionHandler
  );
  public static IEnumerable<IEntry> Search(
    string directory
  , string patterns
  , SearchOptions options
  , int depth
```

```
  , OnProgress progressHandler
  , OnException exceptionHandler
  );
  public static class BreadthFirst
  {
    . . . // Search() x 5 same overloads
  }
  public static class DepthFirst
  {
    . . . // Search() x 5 same overloads
  }
// Utility Operations
  public static IEntry Stat(string path);
  public static long CalculateDirectorySize(string directory);
  public static long CalculateDirectorySize(string directory, int
        depth);
}
```

The first thing to note is that there are 15 search functions, in three groups of five, representing depth-first, breadth-first, and mechanism-agnostic search; each returns an enumerable type implementing the *IEnumerable<IEntry>* interface. It would certainly have been possible to include *BreadthFirst* and *DepthFirst* flags in the *SearchOptions* enumeration, but it's unlikely that a choice between depth-first and breadth-first search is one you will need to make at runtime, and expressing design-time choices at runtime is best avoided because it detracts from discoverability.

Rather than define 15 methods (5 x *Search()*, 5 x *BreadthFirstSearch()*, 5 x *DepthFirstSearch()*) in the same class, they are segregated them into three groups of 5, with the algorithm-specific overloads associated with the nested (static) classes *BreadthFirst* and *DepthFirst*. Obviously, this transgresses the accepted wisdom that class names be nouns, but in this case it's acceptable because it engenders transparency of client code in the form of human-readable statements, as in:

```
foreach(IEntry in FileSearcher.BreadthFirst.Search(@"C:\Windows",
"*.fon|*.ttf"))
{
  Console.WriteLine(entry);
}
```

This is a technique that will be familiar to many .NET programmers. The provision of (static) *Write()/WriteLine()* methods of the *Console* class offers a syntactic convenience over calling them via its *Out* (static) property. Similarly, the notionally algorithm-agnostic *FileSearcher.Search()* methods simply call corresponding *FileSearcher.DepthFirst.Search()* methods. (This is consistent with recls 1.x, where depth-first was the only algorithm.)

We've now discussed the nuances of all parameters, with the exception of depth. There are two special values for depth: *FileSearcher.UnrestrictedDepth* and *0*. The former places no restrictions on depth. The latter causes the search to be non-recursive, i.e. it searches only in the specified directory.

Special Search Functions

As UNIX programmers will know, the *stat()* system call provides status information about a given path, in the form of the struct stat type. The recls core C API provides the function *Recls_Stat()*, which provides status information about a given path, in the form of the *recls_info_t* type (a multi-attribute type analogous to *IEntryj*). Several recls mappings provide a *stat()/Stat()* method that returns a file entry object, or null/nil if no such entry exists. I have found this a handy tool over the years, particularly when working in Python and Ruby, and I wanted to continue to offer it for .NET users, as *FileSearcher.Stat()*. This method either returns null if the file does not exist, or an instance implementing *IEntry* representing the filesystem entry if it can be accessed, or throws an exception if it cannot. (In other words, *System.IO.FileNotFoundException* and *System.IO.DirectoryNotFoundException* are caught, and null returned.)

The other function set, *FileSearcher.CalculateDirectorySize()*, does exactly what it says on the tin: It calculates the size of a directory, as the sum of the sizes of all files in that directory or in any of its subdirectories (up to a given depth). Since this is an expensive operation, I chose not to have directory size automatically calculated during *a b>Search()*-based enumeration. But it's a useful thing to have available, as in the following example, which displays the sizes of all immediate subdirectories of the current directory:

Listing 8: Example using CalculateDirectorySize()

```
foreach(IEntry entry in FileSearcher.Search(
  null, null, SearchOptions.Directories,
  0 // Don't recurse
))
{
  Console.WriteLine("{0} : {1}", entry.Path
        , FileSearcher.CalculateDirectorySize(entry.Path));
}
```

Path Utility Functions

As well as the *FileSearcher* methods, recls 100% .NET provides a number of additional utility functions via the static class *PathUtil* (see Listing 9).

Listing 9: PathUtil class interface

```
public static class PathUtil
{
  public static string DeriveRelativePath(string origin, string
target);
  public static string CanonicalizePath(string path);
  public static string GetAbsolutePath(string path);
  public static string GetDirectoryPath(string path);
  public static string GetFile(string path);
  public static string GetDrive(string path);
}
```

Each of these represents some functionality essential to the proper workings of Recls's searching that is not available in, or corrects defective alternatives in, the CLR's path manipulation facilities:

- *DeriveRelativePath(), CanonicalizePath(),* and *GetDrive()* do not have CLR equivalents
- *GetAbsolutePath()* corrects drive-only UNC paths, i.e. "\\server\share" to append a slash, in the same way that *System.IO.Path.GetFullPath()* does for drive-only volume paths, such as "C:"
- *PathUtil.GetDirectoryPath()* yields the directory path — a recls notion of encapsulating drive (for operating systems that have the concept of a drive) + directory — and corrects the (in my opinion) defective behaviour of *System.IO.Path.GetDirectoryName()*, which returns the empty string when given a root path such as "C:\" or "\\server\share\"
- *PathUtil.GetFile()* yields the file component – file name + extension – of a path and works correctly with UNC paths such as "\\server\share" (for which *System.IO.Path.GetFileName*() returns "share"!)

Extension Methods

With C# 3 comes the ability to enhance the (apparent) operations available on existing types by the use of Extension Methods. I've taken advantage of this for recls 100% .NET by adding the *ForEach*, *Select*, and *Where* methods, as shown in Listing 10. We'll see an example of how these are used (with LINQ) shortly.

Listing 10: Search Extensions

```
public static class SearchExtensions
{
  public static void ForEach(
    this IEnumerable<IEntry> sequence
  , Action<IEntry> action
  )
  {
    foreach(IEntry entry in sequence)
    {
      action(entry);
    }
  }
  public static IEnumerable<TTarget> Select<TTarget>(
    this IEnumerable<IEntry> sequence
  , Func<IEntry, TTarget>    function
  )
  {
    foreach(IEntry entry in sequence)
    {
      yield return function(entry);
    }
  }
  public static IEnumerable<IEntry> Where(
    this IEnumerable<IEntry> sequence
  , Func<IEntry, bool>       predicate
  )
  {
    foreach(IEntry entry in sequence)
    {
      if(predicate(entry))
      {
        yield return entry;
      }
    }
  }
}
```

In C++ terms, this is akin to a partial template specialization, because the extension methods are defined only for *IEnumerable<IEntry>*.

Predicates or Functions?

There was one interesting twist here, with implementing *Where*. Since it requires a predicate — a decision function that returns a Boolean value — I defined it in terms of *System.Predicate*, which is a delegate defined as follows:

```
namespace System
{
  public delegate bool Predicate<T>(T arg);
}
```

That works fine with *IEnumerable<IEntry>*, as in Listing 11.

Listing 11: Use of Extension Methods with Predicate(s)

```
namespace WhereDemo
{
  using Recls;
  using System;
  class WhereDemo
  {
    public static void WhereDemo()
    {
      // with lambda expression
      foreach(IEntry entry in FileSearcher.Search(null, null)
        .Where((e) => e.IsReadOnly))
      {
        Console.WriteLine(entry);
      }
      // with anonymous delegate
      foreach(IEntry entry in FileSearcher.Search(null, null)
        .Where(delegate(IEntry e) { return e.IsReadOnly; }))
      {
        Console.WriteLine(entry);
      }
    }
  }
}
```

However, if we add in a "using *System.Linq*;" statement to the *WhereDemo* namespace, we get a compile error (with some namespace qualifications removed for clarity):

```
error CS0121: The call is ambiguous between the following
methods or properties:
'System.Linq.Enumerable.Where<Recls.IEntry>(IEnumerable<IEntry>,
System.Func<IEntry,bool>)' and
'Recls.SearchExtensions.Where(IEnumerable<IEntry>,
System.Predicate<IEntry>)'
```

What appears to be happening here is that the compiler resolves the lambda expression *(e) => e.IsReadOnly*) (or the equivalent anonymous delegate expression, also shown) to *System.Func<IEntry, bool>*, rather than *System.Predicate<IEntry>*.

```
namespace System
{
  public delegate TResult Func<T, TResult>(T arg);
}
```

Consequently, the two possible *Where* (extension) functions each have one precisely matching argument and one possibly matching argument, hence the ambiguity. This is why I had to implement the recls *Where* extension in terms of *System.Func<IEntry, bool>*, giving two precisely matching arguments, and removing the ambiguity. Obviously, if the C# team ever decide to change the compiler to interpret one-parameter Boolean-returning anonymous delegates / lambda expressions as *System.Predicate<>*, any such "partial specialisations" will be broken, so I'm guessing that'll never happen, and we just need to get used to using *System.Func<T, bool>*, even though a predicate makes more sense.

Test Drive

That's probably enough talk about the design. Let's now take a look at the library in action. We've already seen the Windows Font file search, so now let's look at some of the other simple examples that are included with the recls 100% .NET distribution. (For brevity, I'm going to elide the command-line argument handling and other non-relevant aspects here. Check the distribution for the full program listings.)

FindEmptySubdirectories

This example (Listing 12) finds all the empty, accessible subdirectories of the current directory.

Listing 12: Searching for empty directories.

```
SearchOptions all = SearchOptions.IncludeHidden
                  | SearchOptions.IncludeSystem
                  | SearchOptions.IgnoreInaccessibleNodes;

foreach(IEntry directory in FileSearcher.Search(null, null
                                , SearchOptions.Directories | all))
{
  bool fileFound = false;
  foreach(IEntry file in FileSearcher.Search(directory.Path, null
                                , SearchOptions.Files | all))
  {
```

```
    fileFound = true;
    break;
  }
  if(!fileFound)
  {
    Console.WriteLine(entry);
  }
}
```

ShowImmediateSubdirectoriesTotalSizes

This example shows the total sizes of all immediate subdirectories. It is similar to the one above, but it does not recurse.

```
foreach(IEntry entry in FileSearcher.Search(null, null
  , SearchOptions.Directories, 0))
{
  Console.WriteLine("{0} : {1}", entry
    , FileSearcher.CalculateDirectorySize(entry));
}
```

This is actually a really useful tool when you're trying to find where the space is being consumed on a drive. It can also be written in a single statement:

```
FileSearcher.Search(null, null,
  SearchOptions.Directories, 0
).ForEach((e) => Console.WriteLine("{0} : {1}", e
            , FileSearcher.CalculateDirectorySize(e)));
```

ListInaccessibleDirectories

This example (Listing 13) uses the exception handler to list all the inaccessible subdirectories.

Listing 13: Searching for inaccessible directories.

```
FileSearcher.Search(null, null
  , SearchOptions.Directories |
    SearchOptions.IncludeHidden |
    SearchOptions.IncludeSystem
  , FileSearcher.UnrestrictedDepth,
  (string directory, int depth) =>
  {
    Trace.WriteLine("searching " + directory + " [" + depth +
"]");
    return ProgressHandlerResult.Continue;
  },
  (path, x) =>
  {
    Console.WriteLine("could not access {0}: {1}", path,
x.Message);
    return ExceptionHandlerResult.ConsumeExceptionAndContinue;
  }
).ForEach((e) => e = null);
```

The hidden and system flags are specified to ensure the best chance of running into inaccessible directories. For good measure, I have it perform some rudimentary diagnostic logging by specifying a progress handler that traces the directory and depth. Also, note the curious lambda expression in the *ForEach()* call. This is the best I could think of to give a no-op, since we don't need to do anything with the search results, just have it iterate over all the elements accessible in the *IEnumerable<IEntry>* instance returned from *FileSearcher.Search()*.

When run on my work drive, I get the following output:

```
could not access H:\dev\bin\hidden\inaccessible\: Access to the
path 'H:\dev\bin\hidden\inaccessible' is denied.

could not access H:\dev\bin\hidden\inaccessible\: Access to the
path 'H:\dev\bin\hidden\inaccessible' is denied.

could not access H:\System Volume Information\: Access to the
path 'H:\System Volume Information' is denied.

could not access H:\System Volume Information\: Access to the
path 'H:\System Volume Information' is denied.
```

DirectoryEntryCountFrequencyAnalysis

The final directory-oriented example (Listing 14) lists the number of files contained in each directory.

Listing 14: Directory contents frequency analysis.

```
foreach(IEntry dir in FileSearcher.Search(null, null
  , SearchOptions.Directories))
{
  int n = 0;
  foreach(IEntry file in FileSearcher.Search(dir.Path, null
    , SearchOptions.Files, 0))
  {
    ++n;
  }
  Console.WriteLine("{0} has {1} file(s)",
dir.SearchRelativePath, n);
}
```

FindLargestMatchingFile

This example (Listing 15) finds the largest file matching the given pattern(s). I'm including the full listing to illustrate one way of processing command-line arguments into multi-part patterns. (Please note: It's not the best way of handling command-line arguments, but I didn't want to introduce any more dependencies or complexities into the examples.)

Listing 15: Find largest matching file.

```
static void Main(string[] args)
{
  string directory = null;
  List<string> patterns = new List();
  foreach(string arg in args)
  {
    if(0 != arg.Length && '-' == arg[0])
    {
      switch(arg)
      {
        case "--help":
          ShowUsageAndQuit(0);
          break;
        default:
          Console.Error.WriteLine("FindLargestMatchingFile:
unrecognised argument {0}; use --help for usage", arg);
          break;
      }
    }
    else
    {
      if(null == directory && arg.IndexOfAny(new char[] { '?', '*' })
< 0)
      {
        directory = arg;
      }
      else
      {
        if(arg.IndexOfAny(new char[] { Path.DirectorySeparatorChar,
Path.AltDirectorySeparatorChar } ) >= 0)
        {
          Console.Error.WriteLine("invalid pattern: {0}", arg);
          Environment.Exit(1);
        }
        else
        {
          patterns.Add(arg);
        }
      }
    }
  }
  if(0 == patterns.Count)
  {
    patterns.Add(FileSearcher.WildcardsAll);
  }
  IEntry largest = null;
  foreach(IEntry entry in FileSearcher.Search(directory
    , String.Join("|", patterns.ToArray())
    , SearchOptions.None))
  {
    if(null == largest || largest.Size < entry.Size)
    {
      largest = entry;
    }
  }
  if(null == largest)
  {
```

```
    Console.Out.WriteLine("no matching entries found");
  }
  else
  {
    Console.Out.WriteLine("largest entry is {0}, which is {1} bytes"
      , largest.SearchRelativePath, largest.Size);
  }
}
```

FindCertainSmallExecutables (with LINQ)

This example (Listing 16) finds (the search-relative path of) executable modules that are smaller than 10k and read-only, and uses LINQ.

Listing 16: Find small executables using LINQ.

```
var files = FileSearcher.Search(null, "*.exe|*.dll",
SearchOptions.Files);

var modules = from file in files
              where file.Size < 10240 && file.IsReadOnly
              select file.SearchRelativePath;

foreach(var module in modules)
{
  Console.WriteLine("module: {0}", module);
}
```

StatAFile

The last example (Listing 17) illustrates the use of *Stat()* to elicit information about a single filesystem entity. Once again, I'll show the full listing.

Listing 17: Stat() a file

```
static void Main(string[] args)
{
  string path = Assembly.GetEntryAssembly().Location;
  if(0 != args.Length)
  {
    path = args[0];
  }
  IEntry entry = FileSearcher.Stat(path);
  if(null == entry)
  {
    Console.Error.WriteLine("file not found");
  }
  else
  {
    Console.WriteLine("{0,20}:\t{1}", "Path", entry.Path);
    Console.WriteLine("{0,20}:\t{1}", "SearchRelativePath",
entry.SearchRelativePath);
    Console.WriteLine("{0,20}:\t{1}", "Drive", entry.Drive);
    Console.WriteLine("{0,20}:\t{1}", "DirectoryPath",
entry.DirectoryPath);
    Console.WriteLine("{0,20}:\t{1}", "Directory",
entry.Directory);
    Console.WriteLine("{0,20}:\t{1}", "SearchDirectory",
entry.SearchDirectory);
    Console.WriteLine("{0,20}:\t{1}", "UncDrive",
entry.UncDrive);
    Console.WriteLine("{0,20}:\t{1}", "File", entry.File);
    Console.WriteLine("{0,20}:\t{1}", "FileName",
entry.FileName);
    Console.WriteLine("{0,20}:\t{1}", "FileExtension",
entry.FileExtension);
    Console.WriteLine("{0,20}:\t{1}", "CreationTime",
entry.CreationTime);
    Console.WriteLine("{0,20}:\t{1}", "ModificationTime",
entry.ModificationTime);
    Console.WriteLine("{0,20}:\t{1}", "LastAccessTime",
entry.LastAccessTime);
    Console.WriteLine("{0,20}:\t{1}", "LastStatusChangeTime",
entry.LastStatusChangeTime);
    Console.WriteLine("{0,20}:\t{1}", "Size", entry.Size);
    Console.WriteLine("{0,20}:\t{1}", "Attributes",
entry.Attributes);
    Console.WriteLine("{0,20}:\t{1}", "IsReadOnly",
entry.IsReadOnly);
    Console.WriteLine("{0,20}:\t{1}", "IsDirectory",
entry.IsDirectory);
    Console.WriteLine("{0,20}:\t{1}", "IsUnc", entry.IsUnc);
    Console.WriteLine("{0,20}:\t[{1}]", "DirectoryParts",
String.Join(", ", entry.DirectoryParts.ToArray())); // Assumes
"using System.Linq"
  }
}
```

When run on my development system, I get the following output:

```
                Path:
H:\freelibs\recls\100\recls.net\examples\StatASolutionFile\bin\De
bug\StatASolutionFile.exe
  SearchRelativePath:   StatASolutionFile.exe
               Drive:   H:
       DirectoryPath:
H:\freelibs\recls\100\recls.net\examples\StatASolutionFile\bin\De
bug\
           Directory:
\freelibs\recls\100\recls.net\examples\StatASolutionFile\bin\Debu
g\
     SearchDirectory:
H:\freelibs\recls\100\recls.net\examples\StatASolutionFile\bin\De
bug\
            UncDrive:
                File:   StatASolutionFile.exe
            FileName:   StatASolutionFile
       FileExtension:   .exe
        CreationTime:   3/10/2009 6:53:23 AM
    ModificationTime:   3/10/2009 8:05:59 AM
      LastAccessTime:   3/10/2009 8:14:35 AM
LastStatusChangeTime:   3/10/2009 8:05:59 AM
                Size:   7168
          Attributes:   Archive, Compressed
          IsReadOnly:   False
         IsDirectory:   False
               IsUnc:   False
      DirectoryParts:   [\, freelibs\, recls\, 100\, recls.net\,
examples\, StatASolutionFile\, bin\, Debug\]
```

What recls.NET Offers Above .NET's Search Facilities

You may be reading this and thinking "but there have been standard facilities for recursive filesystem search since CLR version 2". And you'd be right. *DirectoryInfo*'s *GetFiles()* and *GetDirectories()* methods have a third overload that takes a parameter of type *System.IO.SearchOption*, which has the enumerators *TopDirectoryOnly* and *AllDirectories*. And it's the same for *Directory*'s *GetFiles()* and *GetDirectories()* methods.

So what does recls 100% .NET provide that is not available in the standard libraries?

- Multi-part patterns. Passing "*.dll|*.exe" to *Directory.GetFiles()* will throw an argument exception, and passing "*.dll;*.exe" will simply return no results.
- Depth-control. With the CLR facilities you have only two choices: no recursion or infinite recursion.
- *IEntry* — so you don't have to ask twice (*DirectoryInfo.GetFiles()/GetDirectories()* do return arrays of *FileInfo/DirectoryInfo*)
- Filtering of hidden & system files. DirectoryInfo.GetFiles() always includes hidden and system files.
- Ignoring inaccessible directories. An *UnauthorizedAccessException* is thrown by *DirectoryInfo.GetFiles()* when asking for all files for a directory that contains an inaccessible subdirectory (like the "System Volume Information" directory sitting in the root level on each local hard drive). Recls allows you to handle this, by specifying handler (via delegate or interface), or to skip any such items via specifying *SearchOptions.IgnoreInaccessibleNodes*.
- Search-relative path. When I'm writing non-trivial filesystem manipulation programs — such as recursive comparisons, recursive copying, and so on — I find the *SearchRelativePath* invaluable. Writing such programs without it would be burdensome, to say the least.

The Future

As mentioned earlier, recls 100% .NET does not currently provide FTP searching. That's something that might be added in a

later version, though at this stage it looks doubtful. Without a commercial imperative to do so, it's likely to languish at the end of one of my long to-do lists.

Also, the new version does not support the specification of multiple patterns where one or more includes a subdirectory, as in:

```
FileSearcher.Search(@"C:\Windows",
"system/*.dll|system32/*.dll");
```

This will be added in a future version.

I am considering a future facility to treat the patterns parameter as a regular expression, which would probably be indicated by a new *SearchOptions* flag. (The main reason I haven't yet is I'm still in two minds about whether (and how) to handle multiple patterns in that form. I'm definitely interested in opinions from users/readers on the subject.)

Finally, other languages will be getting the recls 100% treatment, probably starting with Ruby or Python next year.

Obtaining recls 100% .NET

recls 100% .NET is available from http://recls.net. The download includes the library (which incorporates all the core functionality discussed in this article) along with documentation (Intellisense XML and CHM), and example projects.

Acknowledgments

I'd like to thank my .NET posse — Chris Oldwood, Garth Lancaster, John O'Halloran, and Joy Chan — for their assistance in keeping me to the point and making it interesting. Any failures are my own fault for inadequately addressing their concerns.

References

[1] The recls project; http://recls.org/

[2] http://en.wikipedia.org/wiki/FxCop

[3] "Introducing recls", Matthew Wilson, *C/C++ Users Journal*, November 2003.

[4] *Extended STL, volume 1: Collections and Iterators,* Matthew Wilson, Addison-Wesley, 2007.

[5] "Quality Matters, Part 1: Introductions, and Nomenclature, "Matthew Wilson, *Overload* 92, August 2009.

[6] *Code Complete, 2nd Edition,* Steve McConnell, Microsoft Press, 2004.

[7] "An Enhanced ostream_iterator," Matthew Wilson, *Dr. Dobb's Journal,* June 2007.

[8] *C# In Depth*, Jon Skeet, Manning, 2008

[9] *More Effective C#,* Bill Wagner, Addison-Wesley, 2009.

— Matthew Wilson is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, specializing in C++ and C#/.NET. He is the author of the books Imperfect C++ and Extended STL, a columnist for ACCU, and a resident guru at Dr. Dobb's CodeTalk, focusing on Windows technologies. He can be contacted at matthew@synesis.com.au.

Return to Table of Contents

The One Instruction Wonder

One-Der — a 32-bit TTA CPU that operates at roughly 10 MIPS (and why you might want to use it!)

by Al Williams

One of the original "Star Trek" television episodes involves patients at a facility for the criminally insane. One of the inmates quotes some lines from Shakespeare and announces that she wrote it yesterday. Another character tells her that it had been written by the bard in the past. The woman replies, "Which does not alter the fact that I wrote it again yesterday!"

I suppose in the computer industry it is particularly difficult these days to have a truly original idea, even if you arrive at your idea independent of prior work. I had that experience several years ago. I had just finished a 16-bit CPU design based loosely on Caxton Foster's Blue machine (http://www.youtube.com/watch?v=dt4zezZP8w8&feature=video_response) in his excellent (albeit dated) book *Computer Architecture* (http://www.amazon.com/Computer-Architecture-Science-Caxton-Foster/dp/0442272197). (Yes, I do have strange hobbies.) Like Foster's original, my machine has what I think of as a 1970's minicomputer architecture — its very similar to a DEC or DG or HP machine from that era. I was contemplating starting a new project using some sort of RISC (Reduced Instruction Set Computer) architecture. RISC's advantages are well known. Simplifying the CPU core by reducing the complexity of the instruction set allows faster speeds, more registers, and pipelining to provide the appearance of single cycle execution. RISC has been so popular that even your PC today probably uses a RISC core that is emulating a non-RISC processor!

So I thought if the "R" in RISC is for reduced, how far can you reduce the instruction set of a computer and still make it do useful work? I realized that you could, in fact, make a perfectly functional computer with only a single simple instruction. I drew up several instruction set architectures and became very enamored of the idea. However, a search on the Internet showed that I wasn't the first person to have this realization. Although they aren't extremely common, OISC (short for "One Instruction Set Computers") have been proposed and built before. The type I had designed was known as a TTA (Transfer Triggered Architecture). There have been a few academic designs using this architecture as well as at least one commercial microcontroller (the Maxim MAXQ).

However, I still did not see some of the design features I wanted to explore in any existing implementation so I pressed ahead with the design of my CPU, the One-Der (with apologies to Tom Hanks). In this article, I describe One-Der — a 32-bit TTA CPU that operates at roughly 10 MIPS. I also show you how and why you might actually want to use it. The complete source code and related files are available at http://i.cmpnet.com/ddj/images/article/2009/code/OneDer.zip.

What's the One Instruction?

When I tell this story in person, people are usually squirming with the inevitable question: What's the one instruction? It turns out there's several ways to construct a single instruction CPU, but the method I had stumbled on does everything via a move instruction (hence the name, "Transfer Triggered Architecture").

In a way, it is almost object oriented. Instead of normal instructions, you have functional units that expose addressable registers. Moving information into a "register" causes the functional unit to change. So as a simple example, consider a

math functional unit — what would pass for the accumulator on a normal processor. The functional unit might expose three registers. Register 0 would allow you to read and write data to the accumulator. Writing to register 1 would cause the unit to add the value you wrote to the accumulator while register 2 would subtract the value.

That's an overly simple example, but by building complex functional units you can build up an "instruction set" that does anything you want. Meanwhile, the basic CPU architecture remains very simple and stable. It simply orchestrates gating the contents of a register on the bus and then gating the bus into another register.

Fundamental Architecture and Advantages

In practice, One-Der is a bit more complex, but not much. The main CPU is about 140 lines of Verilog code — less if you deduct the comments. However, this main portion is really just defining the bus between functional units. All the real work is in one of the functional units — even the program memory and the program counter are functional units.

The main bus has two other functions that augment the transfer of data between functional units. First, it allows for conditional transfers. That is, based on a set of conditional tests, a transfer may not be allowed and the bus cycle becomes a NOP instruction.

The second augmentation is the ability to load constants. The program counter functional unit has a special operation: load the next 32-bit word into a special constant register (which is also part of the PC's functional unit). Because many constants don't require the full 32-bit word, there is also a way to use illegal conditional instructions to load shorter (28-bit) constants in a single 32-bit bus cycle (using the same constant register). In addition, one or more "constant" functional units can make common constants (like zero, for example) easily accessible to any instruction.

This points out one of the major advantages of the One-Der architecture. Need a special set of constants that I don't use? It is trivial to add a new functional unit with your constant. A practical functional unit might be more difficult, but the interface of your logic to the CPU is trivial. Each functional unit has an 8-bit address, so you simply need to create a module for your functional unit and instantiate it in the CPU at an unused address. By convention, all the functional units have the same module signature to a point and then custom parameters follow the standard ones. It is often simpler to test individual functional units separately before integrating them with the main CPU, another advantage.

For example, Listing 1 shows the roughly 70 lines of Verilog that make up the current constant unit. The address is one of the parameters passed in when you instantiate the module.

Listing 1

```
'timescale 1ns / 1ps
'default_nettype none
/*****************************************************************
******
One-Der Copyright 2006, 2007, 2008, 2009
by Al Williams (alw@al-williams.com).


This file is part of One-Der.

One-Der is free software: you can redistribute it and/or modify
it
under the terms of the GNU General Public Licenses as published
by the Free Software Foundation, either version 3 of the
License, or
(at your option) any later version.

One-Der is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY: without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public
License
along with One-Der (see LICENSE.TXT).
If not, see http://www.gnu.org/licenses/.

If a non-GPL license is desired, contact the author.

This is the FU that contains 16 useful constants.

*****************************************************************
******/
module FConstant(input wire xreset,           // reset
                                                    input wire
clk0,                // clock
                                                    input wire
invclk,
                                                    input wire
phase,
                                                    inout wire
[31:0] dbus,         // data
                                                    input wire
[12:0] srcadd,   // source add
                                                    input wire
[12:0] dstadd,   // dest add (not used)
                                                    input wire
[12:0] cfg,         // configured address
                                                    output wire
dta,                       // data transfer ack
                                                    input wire
dtain);               // dta input wire


          wire oe;   // tri state control
          wire [3:0] subunit;
// match addresses
          assign oe=srcadd[7:0]==cfg[7:0]&& srcadd[12]==cfg[12];
          assign subunit=srcadd[11:8];
          assign dta=1'b1;

          reg [31:0] const;
          assign dbus=oe?const:32'bz;

// set up the constants
          always @(subunit)
            begin
            case (subunit)
      4'b0000  :  const=32'b0;
      4'b0001  :  const=32'b1;
                  4'b0010  :  const=32'h2;
      4'b0011  :  const=32'h80000000;
                  4'b0100  :  const=32'h4;
                  4'b0101  :  const=32'hff;
                  4'b0110  :  const=32'hff00;
                  4'b0111  :  const=32'hff0000;
                  4'b1000  :  const=32'hff000000;
                  4'b1001  :  const=32'h00000010;
                  4'b1010  :  const=32'ha;
                  4'b1011  :  const=32'hf;
                  4'b1100  :  const=32'hf0;
                  4'b1101  :  const=32'h80;
      4'b1110  :  const=32'haaaaaaaa;
      4'b1111  :  const=32'hffffffff;
                  default:  const=32'h0;
      endcase
                  end

endmodule
```

To add a new set of constants, simply copy and paste this module to a new Verilog file, rename the module, and instantiate it with a new address in the main CPU file (oneder.v) and you can put 16 more constants in your new custom instruction set. For example, here's the line that includes the existing unit:

```
// Constant source (unit 0)
     FConstant constant(xreset, clk0,invclk,phase, dbus, src,
        dst, 13'h0, dta, dta);
```

Constants are a simple example, but perhaps your program would benefit from three accumulators or four different loop counters. No problem. Just instantiate the existing module multiple times, each with a different address. It is just as easy to remove modules to save space on the FPGA if you find your program never uses a particular module.

While it is handy to be able to add, multiply, and subtract functional units by hand, a ripe area of research is using this architectural style in reconfigurable computing applications. A compiler could determine the optimum number and types of functional units and literally build a custom CPU on the fly optimized for the task at hand.

There is one other feature that functional units can use to manipulate the bus. The bus manages a line known as dta (data transfer acknowledge). This bit allows either the sender or the receiver to stall the bus if necessary. This can be useful for slow external memory access for example. The FIO functional unit takes advantage of the bus stall to implement a delay subfunction.

In addition to functional units, One-Der has a set of registers that can act as the source or destination of a transfer. These act like registers on any garden-variety CPU; they simply store values until needed.

Instruction Set Basics

As you might expect, the instruction set is very simple. The way it is laid out it is very easy to read and write the hex op codes directly (although I have a cross assembler I'll tell you about shortly). Figure 1 shows the basic format. Note that the functional units have an 8-bit address and a 4-bit subfunction code.

So an instruction that doesn't use registers and has no conditions looks like:

```
00 sSS dDD
```

Where *SS* is the source unit, s is the source subunit, and the *D*'s represent the destination unit and subunit. Constants are easy to identify also. The short form takes advantage of the fact that if the condition mask is zero, there's no meaningful reason to have the condition match bits set. So a short constant will have a 1 in the first position and you simply have to mask the top bit off. So 1000003F is the constant 3F hex (note that the constants are sign extended to 32-bits so 18000000 is the constant F8000000). Full 32-bit constants start with 00000F01 followed by the full 32-bit constant (this requires two cycles to execute and does not respect condition codes).

Of course, the real meat to this processor isn't the format of the instructions, it is the possible values for the source and destination. Table 1 shows the standard functional units and their addresses. Note the constant functional unit is unit zero, which allows for some mnemonic subfunctions. So 00000001 (source unit 0, subunit 0) loads a zero into the program counter while 00A00001 loads a constant *A*. Of course, that only goes so far and some of the subfunctions are just arbitrarily assigned.

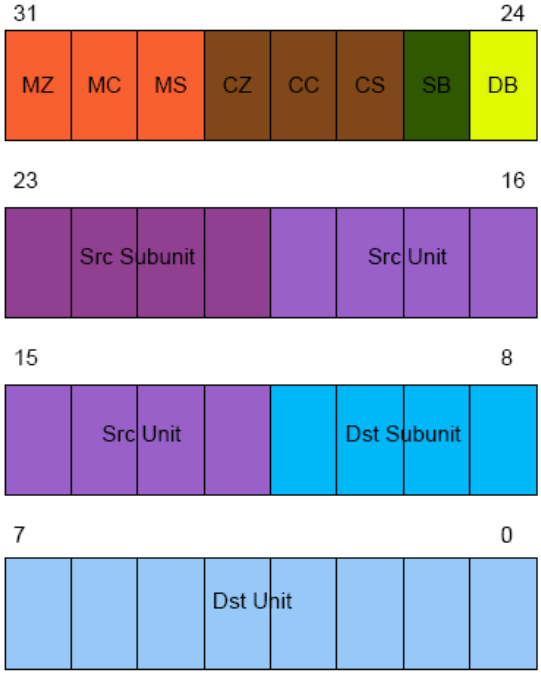

Figure 1: Instruction Format.

Key:
MZ – Condition mask, zero
MC – Condition mask, carry
MS – Condition mask, sign
CZ – Condition state, zero
CC – Condition state, carry
CS – Condition state, sign
SB – Source bank (0=functional unit; 1=register)
DB – Destination bank (0=functional unit; 1=register)
Src Subunit (4 bits) – Subunit for source
Src Unit (8 bits) – Source functional unit
Dst Subunit (4 bits) – Subunit for destination
Dst Unit (8 bits) – Destination functional unit

Example:
80404001 – if not zero (80) move the top of the stack (404) to the program counter (001). Since unit 404 pops the stack, this is effectively a "return on no zero instruction.

Notes:
•If MZ, MC, and MS are zero while CZ=1 then bits 27-0 are treated as a constant that is sign extended to 32-bits and stored in unit E01.

•When the source or destination is a register, the entire source or destination field (12 bits) is available for the register number (although the current implementation only provides 64 registers).

Function	Key Subfunctions	Description
0 - Constant (FCON)	Various constants	Read only -- place constants on the bus
1 - PC (FPC)	PC, add 1, add2, add, immediate, read and write from flash	Program counter; Reads program bytes and embedded constants
2 - Accumulator (FACC)	Add, subtract, shifts, logical operations	Accumulator; condition codes are tied to this function unit
3 - Memory (FMEM)	Two pointers, increment size, value with pre and post increment	Interface to memory (currently uses memory emulated on FPGA)
4 - Stack (FSTK)	push, pop	Manages stack in same memory array as FMEM
5 - Compare (FCMP)	X, target address, logical tests	Tests X against accumulator and returns next PC value if not true, or target value if true
6 - Byte (FBYTE)	Byte and word manipulations	Provides access to accumulator bytes and 16-bit words
7 - Loop (FLOOP)	Two loop counters, start, end, increment, loop address	Sets up two loops; address sent to PC depends on if loop terminated or not
8 - Aux Accumulator (FACC2)	See FACC	Second instance of accumulator; does not alter condition flags
FE - I/O (FIO)	LEDs, switches, UART, delay, general-purpose I/O	Provides I/O capabilities including 7-segment LED decoding and RS232
FF - Reserved	Used internally	

Table 1. Default Functional Units.

Although every instruction is technically a move, you might prefer to give some mnemonic aliases to special moves (my cross assembler does this). For example, moving something into the program counter is a jump. A subroutine return involves moving the top of the stack to the program counter.

CPU Tools

You might be wondering: This all sounds interesting, but how can I implement a custom CPU? The answer is to use a Field Programmable Gate Array (FPGA) device. (For more on programmable logic devices like FPGAs, see my article "Programmable Logic and Hardware" at http://www.ddj.com/architect/184405342.)

In a nutshell, an FPGA implements a large number of logic cells and a programmable way to interconnect them. You can create your design by drawing schematics (only feasible for small designs) or by using a hardware description language (like Verilog, the language used to implement One-Der). A program running on a PC takes your description and converts it into a configuration of the FPGA's logic cells. The result is downloaded to the FPGA by the PC's parallel port. You can also download the result to an EEPROM which can automatically configure the FPGA on startup once you are happy with your design.

Working with FPGAs used to be very expensive. Today, vendors offer development boards that work with free software for well under US$100. My prototype of the One-Der architecture works on a development board available from Digilent (see Resources) that has the equivalent of about 1,000,000 logic gates (it costs a bit more than $100, but was well under $200). In addition to the Xilinx Spartan 3 FPGA chip and assorted support functions, the board also contains some memory devices along with a handful of I/O devices such as switches, LEDs, and a serial port. Of course, none of these things do anything unless your logic makes them do something. The One-Der prototype can read the switches, light the LEDs, and implements a serial port you can use in your programs. It also provides 16 bits of general-purpose outputs and another 16 bits of input available on the board's edge connector.

Verilog

If you aren't familiar with Verilog, you'll notice it strongly resembles C. However, the way it is handled is very different since the result isn't object code, it's a hardware configuration. Consider the following C snippet:

```
x=a|b|~c;
y=a&b;
```

This actually generates code that sets *x* at some discrete time, then sets *y* a short time later. After this code executes, *x* and *y* won't change unless the program reexecutes this code, or executes some other code that modifies *x* and *y*.

In Verilog you might write:

```
assign x=a|b|~c;
assign y=a&b;
```

This will create several logic gates; one that drives the *x* output and one that drives the *y* output. These gates will constantly compute the equations in hardware and the value of *x* (or *y*) at any instant will reflect the current values of the inputs. There's no sequence of computations implied by the line order. The resulting logic is all in parallel.

You can specify gates directly or use other methods, but I find them all clunky compared to using *assign*. For completeness though, here's two other ways you could drive the *x* output equivalently:

```
or(x,a,b,~c);

always @(a,b,c)
  x=a|b|~c;
```

All of these build asynchronous logic in Verilog. You can also create synchronous logic which make up the bulk of the CPU. With synchronous logic, everything is referenced to a clock pulse. This way outputs have until the next clock pulse to "settle" and you can build very complex logic without worrying too much about the different delays encountered by the various signals.

For example, consider generating parity on a serial bit stream. The idea is to use a flip flop to remember the current parity value. When the serial data arrives (at the rising edge of the clock) the new parity will be the old parity exclusive-ored with the data bit. Here's a Verilog module to implement this logic:

```
module serialparity(input clk, input reset, input data, output
reg even, output odd)

assign odd=~even;
always @(posedge clk)
   if (reset) even=1'b0; else even=even^data;
endmodule
```

This defines a module (like a parallel subroutine) that "executes" on each positive clock edge. If you read C, you can probably puzzle out most of the syntax. The *1'b0* means a literal zero that is 1-bit long (the *b* is for binary). In English, the module takes each data bit and exclusive or's it with the previous parity value to form the next parity value.

Of course, a complete Verilog tutorial is beyond the scope of this article, but there's plenty of resources on the Internet. The CPU is relatively simple, but it probably isn't the ideal first Verilog project. On the other hand, adding functional units and manipulating existing ones is well within the reach of the Verilog neophyte

Tool Chain

Assuming you have a bunch of Verilog, how do you get it to the hardware? First, most designs will require more than just straight Verilog. You also need constraints which tell the software how to handle certain parts of the Verilog. For example, the parts of One-Der that read the push buttons or drive the LEDs must connect to certain external pins on the FPGA. Without constraints, the translator software will assign them to arbitrary pins which could cause results ranging from things not working, to physically damaging the device by shorting out something on the board.

There are several ways to include constraints. You can put special comments in your Verilog, but I prefer to use a constraint file. The Xilinx tools (http://www.xilinx.com/) even include a GUI editor for constraint files to simplify the task.

There are other files you may need. The Xilinx tools have a variety of wizards and code generators that allow you to use GUIs to generate special functions (for example, memory or clock circuits). These files will also be a part of your project.

Xilinx provides several versions of their ISE software (http://www.xilinx.com/tools/designtools.htm), which is what you'll use to convert your project into a file that you can download to the FPGA. One version — WebPack (http://www.xilinx.com/support/documentation/boards_and_kits/xtp050.pdf) — is available for free and is all you'll need for many projects. The tool performs several steps to arrive at the final product. First, it does synthesis, mapping your logic descriptions into primitive logic. When this is successful, that logic is mapped to the available FPGA structures. Finally, the structures are arranged on the FPGA and the tool selects how the structures will be interconnected (a process called place and route). Each logic block and interconnect has a certain delay associated with it, so the tool may have to try many different possibilities to arrive at an arrangement that will meet your desired clock speed (the clock speed is specified as a constraint and arriving at a workable solution is known as "meeting timing." For a complex design like One-Der, these steps can take quite some time, even on a fast PC. Once place and route completes, the tools generate a bit file — an image that you can download to the FPGA using the Xilinx-provided tools (usually using a JTAG cable connected to a printer port or a USB port).

To map these operations to the typical software development flow, you can consider ISE a compiler and linker. The synthesis is roughly analogous to compiling source code to assembly language and the mapping is a little like an assembler converting to binary codes. The place and route step is somewhat like linking everything together, although that analogy doesn't cover everything since a real linker can just put all the output objects in any order. The FPGA router has to use the limited interconnects to get everything that needs to communicate connected.

For smaller designs, you can also use ISE to simulate your work without downloading to the actual IC. In fact, I often do this with small parts of One-Der, but the entire CPU is so complex it is difficult (and slow) to simulate well so I usually try to debug directly on the hardware (although simulating an individual functional unit is relatively easy). Xilinx also provides a tool (but not for free) called ChipScope that allows you to perform basic logic analysis by building a dedicated logic analyzer into the FPGA alongside your design. The analyzer accepts commands and sends data to your PC via the FPGA's JTAG port (the same port you use to program the FPGA during development).

For creating One-Der programs I have a simple cross assembler that uses a combination of awk, and the system's C compiler to output binary in several formats. The mem2flash script (available at "http://i.cmpnet.com/ddj/images/article/2009/code/OneDer.zip) lets you load binary from the assembler directly to the chip's on board flash without having to rebuild the processor just to get a new program loaded.

Road Map

The complete source code for One-Der is available at http://i.cmpnet.com/ddj/images/article/2009/code/OneDer.zip. Here's a road map to the source code and a summary of what you'll find in each one:

- topbox.v. This is what connects One-Der to the outside world. I tried to make sure nearly all the board-specific and Spartan 3 specific items show up in this file. The CPU originally used a single clock (the bus would charge on the rising edge and latch on the falling edge) but this made it difficult to do certain memory accesses and limited the CPU speed unnecessarily. After several experiments, I finally decided on using two clock signals, one the inverse of the other and have each instruction take two cycles. The first cycle essentially decodes the instruction while the second one. This file also handles providing a reset signal to the CPU.
- topbox.ucf. The constraints for topbox make certain the switches and LEDs are all connected to the right pins for the Digilent board. Note that PB3 is the reset switch and not used for I/O.
- newclock.xaw. Automatically generated from the clocking wizard, this file provides the ganged DCMs used in topbox.
- oneder.v. This is the main CPU code that implements the bus.
- program.v. This simple file decodes memory addresses and provides access to bflash or rom depending on the address.
- rom.v. This automatically generated file represents a small boot ROM (One-Der boots to ROM placed near the top of memory). The change the contents, edit rom.coe and regenerate the rom core.
- bflash.v. Another automatically generated file. This is the emulated flash where One-Der gets its program code. Note to change the program you need to change the program.coe file

and then manually regenerate the bflash core (the Xilinx tools do not detect that the program.coe file changed). However, the mem2flash script can replace the flash contents on the fly without requiring a rebuild of the CPU.
- FPC.v. The program counter functional unit.
- FIO.v, Debouncer.v, DisplayHex,v, uart.v, rcvr.v, txmit.v. The I/O functional unit and its components.
- FMem.v, FMemStack.v, dualmem.v. These modules provide access to memory (provided by dualmem — an automatically generated dual port memory on the FPGA). Both FMem.v and FMemStack.v share the single memory array. Note that the stack snoops the bus during a subroutine call to save the return address.
- FAcc.v, cmp.v FByte.v, FLoop.v, Fconstant.v. Other functional units.
- FReg.v. The register array. Note that register 0 is used to indirectly address other registers. Reading or writing register 0 actually reads or writes the register pointed to by register 1.
- bflash.bmm. This file describes the emulated flash on the CPU so the mem2flash script can merge flash memory contents with the existing .bit file to produce a new CPU image without rebuilding everything.

More About Units

There are many ways to solve any problem using the One-Der architecture. For example, conditional bus transfers allow you to make decisions. But functional units might use other methods. For example, the loop unit requires you to move a subunit into the program counter. The value of the subunit will vary depending on if the loop is complete or not.

Each unit receives a clock signal and the inverted clock (useful for the Xilinx stock memory which always looks for a rising edge). There is also an alternating phase signal that provides two distinct clock pulses (four clock edges, since the CPU works on both edges of the clock). The clock cycles have the following purposes:

- Clock A: rising edge, phase=0 — Latch instruction from program memory, early decode
- Clock B: falling edge, phase=0 — Set up memory addresses for reading, late decoding
- Clock C: rising edge, phase=1 — Place source data on bus
- Clock D: falling edge, phase=1 — Place bus into destination, update program counter, etc.

The CPU uses program information in clock *B* for some units (mostly those requiring time to set up memory transfers) and in clock *C* for others. This isn't strictly necessary, but it helps the Xilinx tools perform better since there s more time for signals to propagate for most functional units.

Functional units can be simple or complex. If a functional unit needs more than one bus cycle you can easily stall the bus by asserting the dta line in the functional unit. You can implement bus stalls in several ways. For example, suppose you develop a pipelined multiplier functional unit that requires 64 clock cycles to operate. One approach would be to stall the bus when the final operand is sent to the unit. Or you might prefer to let the unit calculate and only stall the bus if the program attempts to read the result before the pipeline completes.

The Future

Listing 2 shows a simple monitor program that allows direct exploration of the One-Der architecture. Of course, you have to share resources with the monitor software. The monitor lets you manually key in small programs or manually set and read functional units.

Listing 2

```
   ;; Simple "monitor" for One-Der
   ;; Williams 1 March 2009

   ;;  assume default stack is ok
   ORG 0

   ;; Registers used for fast subroutine calls
##define CRLF 60
##define HEXO8 61
##define USEND 62
##define READHEX 63
   ;; data registers
##define HEXNUM 59


   ;; Put something on the display so we know we are running
   LDRIQ   0x1234,FIO_DISP
   ;;  init common subroutine calls
   LDRIQ crlf,R(CRLF)
   LDRIQ hexout8,R(HEXO8)
   LDRIQ uartsend,R(USEND)
   LDRIQ readhex,R(READHEX)
   ;; Print a string to the terminal (kill this for more room)
   CALLQ   banner

top:
   LDRIQ '?', FACC     ; prompt
   CALLR USEND
   LDRIQ ' ', FACC     ; note that call clobbers FIMM
   CALLR USEND
   CALLQ uartrx   ; get a character
   CALLR USEND ; echo
   ;; we know about several commands
   ;; fu:
   ;; p x - print x
   ;; s xy - set y=x (note no spaces so s 0felfe for example)
   ;; c n y - set y=n (constant)
   ;; reg:
   ;; d r - print r
   ;; r n r  - set r=n (constant)
   ;; f add data...<esc> - Write to flash  (working)
   ;; g add - Call address
   ;; v add count - view flash add
   ;; you can backspace in a number as long as you haven't
   ;; entered a character <'0' yet

   ;; Vector for each command
   ;; check for p
   MOV FACC,FACC2
   LDRIQ 'p',FACC_SUB
   LDIQ print
   MOVZ FIMMV,FPC
   ;; check for v
   MOV FACC2,FACC
   LDRIQ 'v',FACC_SUB
   LDIQ view
   MOVZ FIMMV,FPC
   ;; check for f
   MOV FACC2, FACC
   LDRIQ 'f',FACC_SUB
   LDIQ flash
   MOVZ FIMMV,FPC
   ;; check for s
   MOV FACC2, FACC
   LDRIQ 's',FACC_SUB
   LDIQ setfu
   MOVZ FIMMV,FPC
   ;; check for c
   MOV FACC2, FACC
   LDRIQ 'c', FACC_SUB
   LDIQ setfucon
   MOVZ FIMMV,FPC
   ;; check for d
   MOV FACC2, FACC
   LDRIQ 'd', FACC_SUB
   LDIQ dispreg
   MOVZ FIMMV,FPC
   ;; check for r
   MOV FACC2, FACC
   LDRIQ 'r', FACC_SUB
   LDIQ setreg
   MOVZ FIMMV,FPC
   ;; check for g
   MOV FACC2, FACC
   LDRIQ 'g', FACC_SUB
   LDIQ go
   MOVZ FIMMV,FPC
```

```
   ;; not found
   JMPQ top

   ;; View flash
view: CALLR READHEX    ; read address
   MOV FACC2, FPC_PGMRDADD
   CALLR READHEX       ; read count
   ;; Set up loop
   MOV FACC2, FLOOP_IEND
   MOV FZERO, FLOOP_I
   MOV FONE, FLOOP_IINC
   LDRIQ viewloop, FLOOP_IADD
viewloop:
   MOV FZERO, FPC_PGMRD          ; read flash
   MOV FIMMV, FACC     ; print
   CALLR HEXO8
   LDRIQ ' ', FACC     ; space between words
   CALLR USEND
   MOV FLOOP_IADD,FPC  ; loop
   CALLR CRLF     ; done
   JMPQ top

   ;;  Call a user subroutine
go:   CALLR READHEX    ; get address
   MOV FACC2,FPC_CALL  ; do it
   JMPQ top    ; done

   ;; Set a register to a constant
setreg:
   CALLR READHEX    ; get constant
   PUSH FACC2
   CALLR READHEX    ; get register #
   LDRIQ 0xFFF,FACC2_AND
   LDRIQ 0x1008000,FACC2_OR ; build instruction to exec
   JMPQ setfu0     ; from here out, same as setting a FU


   ;; printa register
dispreg:
   CALLR READHEX    ; which register?
   LDRIQ 12,FACC2_SHL
   LDRIQ 0xFFF000,FACC2_AND       ; get legit part
   LDRIQ 0x2000002, FACC2_OR      ; build instruction
   JMPQ dispexec    ; from here out, same as printing FU

   ;; Set a Functional unit
   ;; Note this just sort of execs one instruction
setfu:   CALLR READHEX             ; get src and dest
                                     (e.g., 0fe002 is switch->acc)
   JMPQ setfu0    ; from here out, same as set FU constant


setfucon:
   CALLR READHEX    ; get constant
   PUSH FACC2
   CALLR READHEX    ; get destination
   LDRIQ 0xFFF,FACC2_AND          ; build instruction
   LDRIQ 0x008000,FACC2_OR
setfu0:          ; execute instruction
   MOV  FACC2,FPC_WDATA
   LDRIQ execute,FPC_WADD
   MOV FZERO,FPC_WRITE
   POP FACC2
   CALLQ execute
   JMPQ top



   ;; Print an FU
print:
   CALLR READHEX    ; which FU?
   MOV   FACC2,FACC ; print fu address:value
   CALLR  HEXO8
   LDRIQ ':',FACC
   CALLR USEND
   LDRIQ 12,FACC2_SHL
   LDRIQ 0xFFF000,FACC2_AND       ; get legit part
   LDRIQ 0x002, FACC2_OR          ; build instruction
dispexec:
   MOV   FACC2,FPC_WDATA          ; execute instruction and print
   LDRIQ execute,FPC_WADD
   MOV   FZERO,FPC_WRITE
   CALLQ  execute
   CALLR HEXO8
   CALLR CRLF
   JMPQ top

   ;; write to flash
flash:
   CALLR READHEX    ; Get address
   MOV FACC2,FPC_WADD
flashloop:
   MOV FPC_WADD,FIO_DISP          ; echo address to LED display
   CALLR READHEX    ; get data value (FACC2 is value FACC is terminator)
   LDRIQ 0x1B,FACC_SUB ; test for Esc
   JMPZQ top
   MOV FACC2,FPC_WDATA ; write data
   MOV FZERO,FPC_WRITE
   JMPQ flashloop   ; and loop

   ;; Several commands have to execute a dynamic instruction
   ;; So the monitor builds a command and places it here
   ;; and then calls execute
execute:       ; this is how we overwrite flash
   MOV  FZERO,FZERO
   RET

   ;; Canned routines for UART, etc.
##include library.inc

   ;; print a stored banner
banner:
   LDRIQ message,FPC_PGMRDADD
bann1:
   MOV FZERO,FPC_PGMRD
   MOV FIMMV,FACC
   RETZ
   CALLR USEND
   JMPQ bann1
message:
   ; note stringpack crams text into words
   ;; while string outputs individual bytes
   STRING "1Mon"
   DATA 13
   DATA 10
   DATA 0

   END
```

What's next for One-Der? There's an endless number of functional units that could be written, of course. In addition, I have plans to add time-based, external, and debugging interrupts. An external memory interface is also in the works. However, I'm close to shifting gears to think more about tools. A simple example would be a "wizard" style interface to select functional units and generate a custom instance of One-Der. I'm even more interested in working on compiler support to automatically select the optimum number and types of units. Perhaps one day, a One-Der savvy compiler would even take advantage of the ability to partially reconfigure the FPGA to change the functional unit distribution on the fly!

Even so, One-Der is imminently usable as it is. Unlike many other FPGA CPU cores, this one is very simple to customize even if you aren't an expert on its internals. Applications that can benefit from custom instruction in hardware — things like digital signal processing, for example — are ideal for One-Der since you can implement parts of your algorithm in hardware and then easily integrate those parts with the CPU.

Resources

Xilinx at http://www.xilinx.com

Digilent at http://www.digilentinc.com (Note: One-Der was developed on a "Spartan 3 Starter Board" with an XS3C1000 device. The default configuration uses an XS3C200.)

— Al Williams straddles the hardware and software line and has authored books ranging from MFC Black Book to Build Your Own Printed Circuit Board.. He can be contacted at al.williams@awce.com.

Return to Table of Contents

A Build System for Complex Projects: Part 5

Testing and extending the ibs build system

by Gigi Sayfan

This is the fifth and last article in a series of articles that explore an innovative build system for complicated projects. The previous articles discussed build systems in general and the internals of the ideal build system that can integrate with existing build systems. Part 1 (http://www.ddj.com/architect/218400678) and Part 2 (http://www.ddj.com/architect/219000123) discussed build systems in general and the internals of the ideal build system that can integrate with existing build systems. Part 3 (http://www.ddj.com/architect/220100417) discussed in detail how the ideal build system works with the NetBeans IDE and can generate its build files. Part 4 (http://www.ddj.com/architect/220900411) did the same for Microsoft Visual Studio. This installment (Part 5) focuses on testing the flexibility of ibs (short for the "Invisible Build System) and how to extend it in response to new requirements. The complete source code and related files are available here.

Kicking the Tires

ibs was deployed and used as the build system for "Hello World - Enterprise Platinum Edition". The development team started to see how it does. The premise of ibs is that the developers will just add or remove files, directories, libraries, and programs and never need to muck around with build files.

Adding a File

The H team is responsible for the "hello" library that consists of two files called hello.hpp and hello.hpp. Here is the hello.cpp file:

```
#include "hello.hpp"
std::string HelloProvider::getHello()
{
  return "hello";
}
```

This file implements the *HelloProvider::GetHello()* method that returns the string "hello". The H team felt that putting all the eggs in one basket makes it difficult for multiple team members to work in parallel. They decided that it makes more sense to use a divide-and-conquer approach. The new design calls for two new functions — *get_he()* and *get_llo()* — thatwill be used by the *getHello()* method. Here is the code for the two new functions that are placed in a file called helpers.cpp with the prototypes in helpers.hpp:

```
#include "helpers.hpp"
std::string get_he()
{
    return "he";
}
std::string get_llo()
{
    return "llo";
}
```

The *getHello()* method in hello.cpp now uses these functions:

```
#include "hello.hpp"
#include "helpers.hpp"

std::string HelloProvider::getHello()
{
  return get_he() + get_llo();
}
```

The new helpers.cpp and helpers.hpp files were added to all the relevant build files automatically by ibs (requires running build_system_generator.py).

Here is the relevant part of the NetBeans configurations.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configurationDescriptor version="45">
  <logicalFolder name="root" displayName="root"
projectFiles="true">
    <logicalFolder name="HeaderFiles"
                   displayName="Header Files"
                   projectFiles="true">
      <itemPath>hello.hpp</itemPath>
      <itemPath>helpers.hpp</itemPath>
    </logicalFolder>
```

```
    <logicalFolder name="ResourceFiles"
                   displayName="Resource Files"
                   projectFiles="true">
    </logicalFolder>
    <logicalFolder name="SourceFiles"
                   displayName="Source Files"
                   projectFiles="true">
      <itemPath>hello.cpp</itemPath>
      <itemPath>helpers.cpp</itemPath>
    </logicalFolder>
    ...
```

All the other build files were also updated and the new files show up in the NetBeans IDE after reloading the project. Bob recommended to the developers that they close the NetBeans project group (or the Visual Studio solution) before running ibs and reopen it after ibs is done to make sure the IDE is up to date.

Removing a File

Removing a file is just as easy: You simply remove unnecessary files from the file system, run ibs, and watch the removed files disappear from all the build files.

Adding a New Library

The H team was proud of its software engineering acumen and shared their divide-and-conquer approach with the W team responsible for the world library. The W team got excited and wanted to pursue a similar approach. However, Isaac (the development manager) wanted to go even further. He noticed that "hello" and "world" share the letters "o" and "l" and proposed a new reusable letters library that will provide functions for getting important letters. This library can be used by the "hello" and "world" libraries to get all the letters they need.

The U team (responsible for developing the utils library) was assigned the task of creating the letters library. The library consisted two files: letters.cpp and letters.hpp. Each letter needed for the hello world application got its own function. Here is the code for letters.cpp (letters.hpp contains the function prototypes):

```
#include "letters.hpp"

std::string get_h() { return "h"; }
std::string get_e() { return "e"; }
std::string get_l() { return "l"; }
std::string get_o() { return "o"; }
std::string get_w() { return "w"; }
std::string get_r() { return "r"; }
std::string get_d() { return "d"; }
```

The H and W teams modified the *getHello()* and *getWorld()* methods to use the new letters library. The H team also got rid of the helpers.cpp and helpers.hpp files that were no longer needed. Here is the code for world.cpp file, which implements the *getWorld()* method:

```
#include "world.hpp"
#include <hw/letters/letters.hpp>
std::string WorldProvider::getWorld()
{
  return get_w() + get_o() +
         get_r() + get_l() + get_d();
}
```

This is a great example of code reuse and the code base is now very flexible. For example, if the project stakeholders decided that all the "o" letters in the system should be uppercase, only the *get_o()* function of the letters library will have to change and all the libraries and applications using it will just need to relink against it.

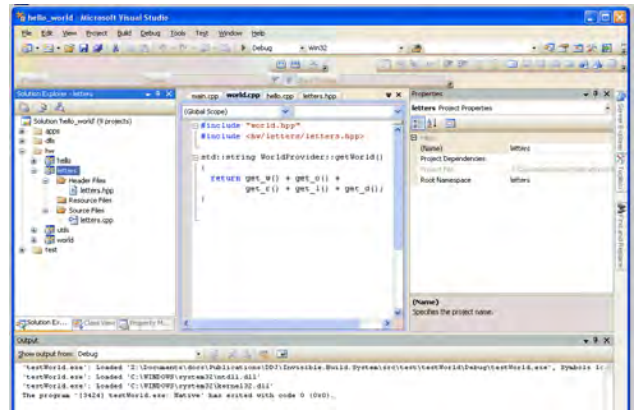

Figure 1.

What kind of changes to the build files are needed to add a new library? First, all the build files necessary to build the library itself, then all the dynamic libraries or executables that depend on it (directly or indirectly) must link against it. In addition, you want to update the workspace file so the new library shows up in the IDE and can be built and debugged in the IDE. Of course, you want to do all that for all the platforms you support. That's a lot of work and it's easy to miss a step or misspell a file here and there. Just figuring out what test programs and applications need to link against the new library is pretty labor intensive. Luckily for Isaac and his development team, ibs can do all that automatically. The single act of placing the letters library under the src\hw directory is enough to tell ibs everything it needs to know. Let's see what ibs did on Windows this time:

- Created the letters.vcproj file in the hw/letters directory.
- Added the letters project to the hello_world solution under the hw folder (see Figure 1).
- Figured out by following the *#include* trail that the "hello" and "world" libraries use "letters" and hence any program that uses either "hello" or "world" depend on "letters" and will link against it automatically. Currently, that's the hello_world application itself and the testHello and testWorld test programs.

Here are the relevant changes to the hello_world.sln file:

```
Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "letters",
"hw\letters\letters.vcproj", "{C27369BC-2E11-4571-B524-
2F0279F202BD}"
EndProject

Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "hello",
"hw\hello\hello.vcproj", "{23B8D8A1-8E84-462B-BF90-58E1F07D267D}"
  ProjectSection(ProjectDependencies) = postProject
        {C27369BC-2E11-4571-B524-2F0279F202BD} = {C27369BC-2E11-
4571-B524-2F0279F202BD}
  EndProjectSection
EndProject

Project("{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}") = "world",
"hw\world\world.vcproj", "{2A5E91EE-8A54-4594-A28E-3185F5F8602C}"
       ProjectSection(ProjectDependencies) = postProject
          {C27369BC-2E11-4571-B524-2F0279F202BD} = {C27369BC-2E11-
4571-B524-2F0279F202BD}
  EndProjectSection
EndProject
```

```
  {C27369BC-2E11-4571-B524-2F0279F202BD}.Debug|Win32.ActiveCfg =
Debug|Win32
  {C27369BC-2E11-4571-B524-2F0279F202BD}.Debug|Win32.Build.0 =
Debug|Win32
  {C27369BC-2E11-4571-B524-2F0279F202BD}.Release|Win32.ActiveCfg
= Release|Win32
  {C27369BC-2E11-4571-B524-2F0279F202BD}.Release|Win32.Build.0 =
Release|Win32

  {C27369BC-2E11-4571-B524-2F0279F202BD} = {0276cb28-8c64-46ae-
9e52-3363bb4dcbd8}
```

Adding a New Test

The U team did a good job with the letters library and to adhere to the development standard, it added a test program too — not TDD (Test Driven Development), but better than no tests at all. The U team created a directory called testLetters under src/test and put the following main.cpp file in it:

```
#include <hw/utils/base.hpp>
#include <hw/letters/letters.hpp>
#include <iostream>

int main(int argc, char** argv)
{
  CHECK(get_h() == std::string("h"));
  CHECK(get_e() == std::string("e"));
  CHECK(get_l() == std::string("l"));
  CHECK(get_o() == std::string("o"));
  CHECK(get_w() == std::string("w"));
  CHECK(get_r() == std::string("r"));
  CHECK(get_d() == std::string("d"));

  return 0;
}
```

After invoking ibs, the new testLetters project became part of the solution and the U team ran the test successfully.

Adding a New Application

The "Hello World – Enterprise Platinum Edition" was a great success and became a killer app overnight. However, some big players weren't satisified with the security of the system and demanded an encrypted version of hello world. Isaac (the development manager) decided that this called for a separate application to keep the original hello_world application nimble and user-friendly. The new application was to be called "Hello Secret World" and print an encrypted version of the string "hello world!". Furthermore, it will not use any of the intensive infrastructure built for the original "Hello World" system. A special no-name clandestine team was recruited to implement it. After a lot of deliberation, the no-name team decided to implement the ultimate encryption algorithm — ROT13 (http://en.wikipedia.org/wiki/ROT13). In addition, the team demonstrated a nice usage of the standard *transform()* algorithm to apply the ROT13 encryption.

```
#include <iostream>
#include <algorithm>

char ROT13(char c)
{
  if (c >= 'a' && c < 'n')
    return char(int(c) + 13);
  else if (c > 'm' && c <= 'z')
    return char(int(c) - 13);
  else
    return c;
}
int main(int argc, char** argv)
{
  std::string s("hello, world!");
  // Apply the secret ROT13 algorithm
  std::transform(s.begin(), s.end(), s.begin(), ROT13);
  std::cout <<s.c_str() << std::endl;
  return 0;
}
```

Again, ibs took care of integrating the new application. The unnamed team just had to put its hello_secret_world application under src/apps.

Extending the Build System

To this point, Bob hasn't made an appearence in this article and it is a good sign. The developers, including the new unnamed team, were able to use ibs effectively without any help from Bob. But, the success of the "hello world" product family brought new demands. Upper management decided that they want to package the "hello world" functionality as a platform and let other developer enjoy "hello world" (for a small fee of course). Isaac conducted a thorough market analysis and concluded that Ruby is the way to go. He summoned Bob and asked him to extend ibs, so it will be possible to provide Ruby bindings for the "hello" and "world" libraries.

Bob started to research the subject, soon discovering that Ruby depends on the gcc toolchain to produce its bindings. It's possible on Windows to generate an NMAKE file for Visual Studio, but Bob decided that he would first take a shot at building a Ruby binding for the Mac OS X only.

Ruby Bindings

A Ruby binding is a dynamic library with a C interface that follows some conventions and uses some special data types and functions from the Ruby C API. The end result is a module that can be consumed by Ruby code.

Here is the C code Bob came up with as a pilot. The "*ruby.h*" header contains the Ruby C API definitions. The *Init_hello_ruby_world()* is the entry point that Ruby calls when it loads the binding. This function defines a class called *HelloWorld* that has two methods called *get_hello()* and *get_world()*. The temporary implementation just returns the strings "hello" and "world". The final version will link to the C++ "Hello, World!" project and utilize its sophisticated services.

```
#include "ruby.h"
static VALUE get_hello(VALUE self)
{
    VALUE result = rb_str_new2("hello");
    return result;
}
static VALUE get_world(VALUE self)
{
    VALUE result = rb_str_new2("world");
    return result;
}
VALUE cHelloWorld;
void Init_hello_ruby_world()
{
    cHelloWorld = rb_define_class("HelloWorld", rb_cObject);
    rb_define_method(cHelloWorld, "get_hello", get_hello, 0);
    rb_define_method(cHelloWorld, "get_world", get_world, 0);
}
```

To make an actual Ruby binding out of this source file, Bob created a Ruby configuration file called extconf.rb that contains just two lines

```
require 'mkmf'
create_makefile("hello_ruby_world")
```

Next, Bob ran the configuration file through Ruby and Ruby generated a Makefile appropriate for the current platform (Mac OS X):

```
~/Invisible.Build.System/src/ruby/hello_ruby_world > ruby extconf.rb
creating Makefile
```

Here is the Makefile:

```
SHELL = /bin/sh

#### Start of system configuration section. ####

srcdir = .
topdir =
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/lib/ru
by/1.8/universal-darwin9.0
hdrdir = $(topdir)
VPATH = $(srcdir):$(topdir):$(hdrdir)
prefix =
$(DESTDIR)/System/Library/Frameworks/Ruby.framework/Versions/1.8/
usr
exec_prefix = $(prefix)
sitedir = $(DESTDIR)/Library/Ruby/Site
rubylibdir = $(libdir)/ruby/$(ruby_version)
docdir = $(datarootdir)/doc/$(PACKAGE)
dvidir = $(docdir)
datarootdir = $(prefix)/share
archdir = $(rubylibdir)/$(arch)
sbindir = $(exec_prefix)/sbin
psdir = $(docdir)
localedir = $(datarootdir)/locale
htmldir = $(docdir)
datadir = $(datarootdir)
includedir = $(prefix)/include
infodir = $(DESTDIR)/usr/share/info
sysconfdir = $(prefix)/etc
mandir = $(DESTDIR)/usr/share/man
libdir = $(exec_prefix)/lib
sharedstatedir = $(prefix)/com
oldincludedir = $(DESTDIR)/usr/include
pdfdir = $(docdir)
sitearchdir = $(sitelibdir)/$(sitearch)
bindir = $(exec_prefix)/bin
localstatedir = $(prefix)/var
sitelibdir = $(sitedir)/$(ruby_version)
libexecdir = $(exec_prefix)/libexec

CC = gcc
LIBRUBY = $(LIBRUBY_SO)
LIBRUBY_A = lib$(RUBY_SO_NAME)-static.a
LIBRUBYARG_SHARED = -l$(RUBY_SO_NAME)
LIBRUBYARG_STATIC = -l$(RUBY_SO_NAME)

RUBY_EXTCONF_H =
CFLAGS   =  -fno-common -arch ppc -arch i386 -Os -pipe -fno-
common
INCFLAGS = -I. -I$(topdir) -I$(hdrdir) -I$(srcdir)
DEFS     =
CPPFLAGS =   $(DEFS)
CXXFLAGS = $(CFLAGS)
DLDFLAGS = -L. -arch ppc -arch i386
LDSHARED = cc -arch ppc -arch i386 -pipe -bundle -undefined
dynamic_lookup
AR = ar
EXEEXT =

RUBY_INSTALL_NAME = ruby
RUBY_SO_NAME = ruby
arch = universal-darwin9.0
sitearch = universal-darwin9.0
ruby_version = 1.8
ruby =
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ru
by
RUBY = $(ruby)
RM = rm -f
MAKEDIRS = mkdir -p
INSTALL = /usr/bin/install -c
INSTALL_PROG = $(INSTALL) -m 0755
INSTALL_DATA = $(INSTALL) -m 644
COPY = cp

#### End of system configuration section. ####

preload =

libpath = . $(libdir)
LIBPATH =  -L. -L$(libdir)

DEFFILE =

CLEANFILES = mkmf.log
DISTCLEANFILES =

extout =
extout_prefix =
target_prefix =
LOCAL_LIBS =
LIBS = $(LIBRUBYARG_SHARED)  -lpthread -ldl -lm -lutils -lhello
-lworld
SRCS = hello_ruby_world.c
OBJS = hello_ruby_world.o
TARGET = hello_ruby_world
DLLIB = $(TARGET).bundle
EXTSTATIC =
STATIC_LIB =

RUBYCOMMONDIR = $(sitedir)$(target_prefix)
RUBYLIBDIR    = $(sitelibdir)$(target_prefix)
RUBYARCHDIR   = $(sitearchdir)$(target_prefix)

TARGET_SO     = $(DLLIB)
CLEANLIBS     = $(TARGET).bundle $(TARGET).il? $(TARGET).tds
$(TARGET).map
CLEANOBJS     = *.o *.a *.s[ol] *.pdb *.exp *.bak

all:  $(DLLIB)
static:   $(STATIC_LIB)

clean:
    @-$(RM) $(CLEANLIBS) $(CLEANOBJS) $(CLEANFILES)

distclean: clean
    @-$(RM) Makefile $(RUBY_EXTCONF_H) conftest.* mkmf.log
    @-$(RM) core ruby$(EXEEXT) *~ $(DISTCLEANFILES)

realclean: distclean
install: install-so install-rb

install-so: $(RUBYARCHDIR)
install-so: $(RUBYARCHDIR)/$(DLLIB)
$(RUBYARCHDIR)/$(DLLIB): $(DLLIB)
  $(INSTALL_PROG) $(DLLIB) $(RUBYARCHDIR)
install-rb: pre-install-rb install-rb-default
install-rb-default: pre-install-rb-default
pre-install-rb: Makefile
pre-install-rb-default: Makefile
$(RUBYARCHDIR):
  $(MAKEDIRS) $@

site-install: site-install-so site-install-rb
site-install-so: install-so
site-install-rb: install-rb

.SUFFIXES: .c .m .cc .cxx .cpp .C .o

.cc.o:
  $(CXX) $(INCFLAGS) $(CPPFLAGS) $(CXXFLAGS) -c $<

.cxx.o:
  $(CXX) $(INCFLAGS) $(CPPFLAGS) $(CXXFLAGS) -c $<

.cpp.o:
  $(CXX) $(INCFLAGS) $(CPPFLAGS) $(CXXFLAGS) -c $<

.C.o:
  $(CXX) $(INCFLAGS) $(CPPFLAGS) $(CXXFLAGS) -c $<

.c.o:
  $(CC) $(INCFLAGS) $(CPPFLAGS) $(CFLAGS) -c $<

$(DLLIB): $(OBJS)
  @-$(RM) $@
  $(LDSHARED) -o $@ $(OBJS) $(LIBPATH) $(DLDFLAGS) $(LOCAL_LIBS)
$(LIBS)

$(OBJS): ruby.h defines.h
```

With a nice Makefile under his belt, Bob proceeded to build the hello_ruby_world binding:

```
~/Invisible.Build.System/src/ruby/hello_ruby_world > make cc -
arch ppc -arch i386 -pipe -bundle -undefined dynamic_lookup -o
hello_ruby_world.bundle hello_ruby_world.o -L. -
L/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/lib -
L. -arch ppc -arch i386 -lruby -lpthread -ldl -lm
```

The result was a hello_ruby_world.bundle file, which is the binding itself (a .dll on Windows, and .so on Linux). Now, Bob invited Isaac to examine the new toy. Isaac, a big Ruby fan, immediately wrote a Ruby test program to make sure the binding is indeed usable from Ruby. The program starts with two *require* statements (the equivalent of *import* in Python). Note that the first one requires the new binding *hello_ruby_world*. Next, it creates a test class that subclasses the standard Ruby *Test::Unit::TestCase* and defines a method that instantiates the *HelloWorld* class from the binding and exercises its methods.

```
require 'hello_ruby_world'
require 'test/unit'

class TestHelloWorld <l Test::Unit::TestCase
  def test_HelloWorld
    hw = HelloWorld.new
    assert_equal(hw.get_hello(), "hello")
    assert_equal(hw.get_world(), "world")
  end
end
```

Isaac executed his test program and was happy with the results:

```
~/Invisible.Build.System/src/ruby/hello_ruby_world > ruby
test_hello_ruby_world.rb

Loaded suite test_hello_ruby_world
Started

Finished in 0.000405 seconds.
```

Isaac also tried the interactive Ruby interpreter (irb):

```
~/Invisible.Build.System/src/ruby/hello_ruby_world > irb
>> require "hello_ruby_world"
=> true
>> hw = HelloWorld.new
=> #<HelloWorld:0x3679b4>
>>  hw.get_hello() + ', ' + hw.get_world() + '!'
=> "hello, world!"
>>
```

Bob was satisfied and it was time to integrate the new capability to generate Ruby bindings into ibs. The proper way to do it was to figure out how to create a NetBeans project and a VisualStudio project that contain the various incantations hidden in the Ruby-generated Makefile. But Bob was pressed for time and the Ruby binding was really needed just for the Max OS X platform. Consequently, Bob decided to utilize Python's agility and integrate the Ruby binding building as a standalone Python program that will have to be invoked by the developers or build master after the build of the C++ projects was over. I'll shortly discuss how to integrate ibs into a full-fledged automated software development lifecycle.

For starters, he created a standalone piece of code to build Ruby extensions. He assumed the following:

- All the Ruby extensions will reside in subdirectories of <root dir>/src/ruby
- The name of the extension will be the name of the directory it resides in
- The developers will write the C extension code

The program he came up with automated the entire process. For each Ruby extension it: generated an extconf.rb configuration file from a template (based on the project path); generated a Makefile from the configuration file; and finally created the extension bundle itself by running 'make'. This code demonstrates one of the simplest ways to invoke external processes like 'ruby' and 'make' from Python using the subprocess module. The *subprocess.call()* function used here doesn't provide a lot of control or interaction with the launched process, but in this case it's enough. The *subprocess* module provides multiple ways to launch and interact with launched processes.

The program is based on the *build_ruby_binding()* function that accepts a project path (the directory that contains the extension's C code) and eventually creates the Ruby bindings bundle in the same directory. The *build_all_ruby_bindings()* function just iterates over all the subdirectories of the src/ruby directory and calls *build_ruby_binding* on each one.

```
import os
import sys
import subprocess

extconf_template = "require 'mkmf'\n create_makefile(\"%s\")"

def build_ruby_binding(project_path):
  """Build a Ruby binding

  - Generate an extconf.rb file (configuration file)
  - Run it through Ruby to generate a Makefile
  - Run the Makefile to build the actual binding
  """
  project_path = os.path.abspath(project_path)
  # Verify the project dir exists
  assert os.path.isdir(project_path)
  name = project_path.split('/')[-1]
  # make sure the binding file exists
  assert os.path.isfile(os.path.join(project_path, name + '.c'))

  save_dir = os.getcwd()
  try:
    os.chdir(project_path)

    # Generate the extconf.rb file
    extconf_rb = extconf_template % name
    open('extconf.rb', 'w').write(extconf_rb)

    # Remove exisitng Makefile
    if os.path.isfile('Makefile'):
      os.remove('Makefile')

    # Invoke the extconf.rb file to generate the Makefile
    subprocess.call(['ruby', 'extconf.rb'])
    assert os.path.isfile('Makefile')

    # Remove exisitng bundle and make a new one
    bundle = name + '.bundle'
    if os.path.isfile(bundle):
      os.remove(bundle)
    subprocess.call(['make'])
    assert os.path.isfile(bundle)

  finally:
    os.chdir(save_dir)

def build_all_ruby_bindings(ruby_dir):
  subdirs = os.walk(ruby_dir).next()[1]
  for s in subdirs:
    build_ruby_binding(s)


if __name__=='__main__':
  ruby_dir = '.'
  build_all_ruby_bindings(ruby_dir)
```

Debugging Build Problems

Sometimes builds fail. There are many possible reasons. With ibs, there could be problems during the generation process or during

the build itself. If the problem happens during the build system generation, then you can just run the build_system_generator.py script in the debugger and put a breakpoint in the problematic area.

> It is important to start small and grow the build system incrementally

One problem that happens a lot with other build systems is missing or misnamed files. This occurs if a file is moved, renamed, or just deleted but the corresponding build file is not updated. With ibs, this can't happen because the build files are generated automatically based on the existing files. But, a source file might reference a missing file. This will be discovered during the build itself.

Another common problem is link failure. That happens if an executable or dynamic library depends on a static library, which is not linked into it. There are two reasons for link failures:

1. The dependency is not specified in the build file
2. The static library failed to build

Failure #1 can't happen with ibs because it detects all dependencies automatically and adds them to the build files. Failure #2 is easy to detect because the static library will fail to build before the executable or dynamic library fail to link.

A more difficult failure that can't be detected automatically is dependency on dynamic library. You must come up with some system to track and manage dynamic library dependencies. You will usually get a clear error message that says that such and such dynamic library can't be loaded.

If you integrated testing into your build system then the most common failures will be test failures, which you just need to fix.

There could be other failures if your build system is even more sophisticated and performs other tasks like compiling documentation, packages your system for deployment, and uploads to a staging area or a web server.

The key is always to prevent as much as you can and make sure that failures are easy to detect with good diagnostic messages that contain all the information needed to correct the problem.

Developing a Custom Build System

ibs, the build system I described in this series, focused on building C/C++ source files in a cross-platform way or rather generating build files. A serious industrial-strength build system does much more. If you want to develop your own build system, you need to consider these aspects. The absolute minimum must include checking out the sources from source control, building all the software artifacts on all platforms, running a test suite on all platforms, and reporting the results.

The automated test suite is a critical piece for a professional software organization and it gets as fancy as you want with a complete test environment that simulate your deployment environment, automated GUI tests and complete builds and tests of your source releases (that's right — building the source is part of the test).

Then there is packaging. There are many ways today to distribute software. You may develop a web application, a native client, a smart phone app, or a plug-in to some other application like Firefox, Eclipse, or Visual Studio. Probably, you end up with multiple artifacts that need to be packaged and deployed. Your build system should take care of this aspect, too.

Automatically generated documentation is also in the realm of the build system. In general, almost any repetitive task can be automated with some imagination.

It is important to start small and grow the build system incrementally. If you try to nail everything down before you let the developers make the first check in you won't get very far.

The best guideline is to address pain points as they show up. If your developers keep having problems with third-party dependencies, then figure out a way to verify it before check in. But if you release software every two years, there is probably no need for automatically creating an installer for fancy GUI client.

Conclusion

In this five-part article series, I delved into the sometimes mysterious world of building software. I described the issues involved in building cross-platform C/C++ code and presented a unique build system called ibs (Invisible Build System) that addresses many of the issues. I explored the design and implementation in great detail. I even tried to be funny by showcasing the build system through the most bloated "Hello World" application I could conceive. I hope you liked this series and that some of you will find it useful and may even try to create your own build system. It's a lot of fun.

— Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

Return to Table of Contents

Go: A New Programming Language from Google

A concurrent systems programming language

by Gastón Hillar

Google has launched "Go," a new systems programming language born with concurrency, simplicity, and performance in mind. Go is open source and its syntax is similar to C, C++ and Python. It uses an expressive language with pointer but no pointer arithmetic. It is type safe and memory safe. However, one of its main goals is to offer the speed and safety of a static language but with the advantages offered by modern dynamic languages. Go also offers methods for any type, closures and run-time reflection. The syntax is pretty clean and it is garbage collected. It is intended to compete with C and C++ as a systems programming language.

What about multicore programming with Go? It promotes lightweight concurrency allowing developers to create sets of lightweight communicating processes. Go calls them goroutines. This way, you can run many concurrent goroutines and you don't need to worry about stack overflows. Go promotes sharing memory by communicating. Goroutines aren't threads, they are functions running in parallel with other goroutines in the same address space. It is very easy to launch parallel functions using the goroutines. This is one of the most interesting features offered by the language. It really simplifies concurrency for systems programming.

Go's key features related to concurrency are:

- Channels.
- Channels of channels.
- Goroutines.
- Leaky buffers.
- Share by communicating approach.

These features deserve new posts explaining them with more detail. Stay tuned because I'll be adding new posts about Go soon.

The idea behind Go is to offer a fast compiler to produce fast code. So far, it offers two compilers:

- Gccgo (GCC is in the back).
- 8g (x86-32) and 6g (x86-64).

If you want to test this new programming language, your starting point is Go's main page at http://golang.org/.

— *Gastón Hillar is the author of* C# 2008 and 2005 Threaded Programming: Beginner's Guide.

Return to Table of Contents

Q&A: Software Testing In a Virtualized World

Do virtualization and cloud computing pose unique testing challenges?

by Jonathan Erickson

James Whittaker is director of test engineering for Google, and author of *How To Break Software: A Practical Guide To Testing*. He recently spoke with *Dr. Dobb's* editor in chief Jonathan Erickson.

Dr. Dobb's: Do virtualization and cloud computing pose unique testing challenges?

Whittaker: Opportunities more than challenges. At Google, if I want to test, say, Chrome, I visit a Web site, tell it how many machines I want and what operating systems, drivers, apps, and the version of Chrome that I want on them, and wham! those machines are provisioned, and I can point my test automation at them. I don't care where they are. I don't care what they are. They exist, and they act just like the test environment that I would otherwise have to painstakingly — and expensively — create.

Dr. Dobb's: What about multicore platforms and parallel programming?

Whittaker: Multicore behaves the same as single core from an external point of view. Same with parallel and serial. The difference is with unit-level and other code-based tests. The devil is in these low-level details, and tools haven't yet caught up.

Dr. Dobb's: Functional testing, unit testing, security testing, and more. What's next?

Whittaker: Accessibility -- hands down. The idea that we can abstract the input mechanics from the functionality of the app. The idea that anything the application is capable of doing can be invoked programmatically. For users with disabilities, this is crucial as it allows for a great deal of creativity in how the program is manipulated. For testers, this means the ultimate set of test hooks. With accessible code, I can write hooks that can literally drive it through its entire set of capabilities. Nothing needs to be left to chance anymore.

Dr. Dobb's:: How close are we to "real" automated testing?

Whittaker: Whatever buttons users can press, whatever values they can enter, we can see and do with automation. But applying inputs is only the first part of manual testing. Human testers can see subtle variations that lead them to say "that's a bug." This is the primary limiting aspect of automation. Programs aren't good at seeing output and processing behavior. They can see crashes. But they can't notice major bugs like navigating to the wrong page or rendering an image incorrectly.

Return to Table of Contents

Professional Ubuntu Mobile Development Book Review

Reviewed by Mike Riley

Professional Ubuntu Mobile Development
Ian Lawrence, Rodrigo Cesar Lopes Belem
Wiley Publishing
$59.99

As the move toward mobile computing continues to progress at an ever-accelerating rate, platforms that began life on the desktop are crossing the chasm to more personal portable designs. Does this book successfully orient desktop developers to this new horizon? Read on to find out.

Written by two software developers from the Insituto Nokia de Tecnologia, *Professional Ubuntu Mobile Development* moves technically capable readers along at a swift pace. After a brief history lesson, the book establishes reasons why application developers need to embrace mobile development trends and why the Linux operating system provides the foundation for future mobile devices. Several more pages than necessary are spent on obtaining, installing and configuring a virtual development environment using VirtualBox, KVM and/or QEMU; establishing the networking configurations are the most annoying aspects of these open source VM hosts. Once the VM's are ready to go, the book hits its stride with its useful analysis of power management, a 50+ page chapter on application development (recommending Hildon, a handheld application framework, and other tools like Canola, Clutter, EFL, Elementary, Glade and Qt), application packaging using Debian's apt (Advanced Package Tool) and Canonical's Launchpad PPA (Personal Package Archives). The book then evaluates the type of applications that best suit the portable constraints of today's mobile technology. Ideas for business, multimedia, social network and location-aware users are demonstrated.

Chapters on Theming (customizing various Ubuntu distribution files in order to alter the appearance of the boot up, login and desktop GUI's), Linux kernel fine-tuning and even testing and usability are covered. A brief chapter on 'tips & tricks' (ex: configuring a touch screen with the 'evtouch' package and monitoring hard drive activity with the 'iotop' application) is followed by a chapter that "puts it all together" by walking readers through "the process of creating a custom distribution of Ubuntu Mobile" from start to finish. "Mobile Directions" revisits the book's mobile trend premise and posits new problems brought about by the vast proliferation of mobile computing devices.

The book's final chapter on mobile development common problems and possible solutions seemed tacked on and out of place, as these topics should have been part of the tips & tricks section. Appendix A relates a fascinating case study about the construction of a SWARM (Sheeva/Solar Powered, Wireless, Advanced/Application, Running, Memcached/Machines) by MIT students using the SheevaPlug (http://www.marvell.com/products/embedded_processors/developer/kirkwood/sheevaplug.jsp) that demonstrates the efficacy of the principles promoted by the book. A solid Git overview, Launchpad project hosting, a Python-based Desktop Power Applet example and a D-Bus overview are covered in Appendixes B through E respectively.

Intermediate and advanced Linux developers won't learn much new reading the book. Even so, after tooling around with the latest Ubuntu Netbook Remix (http://www.ubuntu.com/GetUbuntu/download-netbook), it's easy to understand how custom applications optimized for this platform could greatly benefit taking into account the screen real-estate constraints, reduced power consumption and CPU capacity current Atom processor-based netbooks have to deal with. The thought of dedicating a few weeks on completely customizing my own remix running my own netbook-centric apps and desktop widgets is enticing. Alas, until I am stranded on a desert island with a couple of working, ruggedized netbooks, a solid WiFi Internet connection and a solar array powerful enough to keep the netbooks powered 24x7, I will have to add such a desire to my ever-growing 'things to do' mind map. In the meantime, students and computing professionals with the hardware, connectivity and available time to tinker will find *Professional Ubuntu Mobile Development* enough of a catalyst to kickstart the implementation of their own mobile-oriented visions.

Return to Table of Contents

Staying On Track While Everything Changes

How developer managers can cope with the ever-shifting requirements and moving deadlines of today's projects

by Mike Shepherd

Change, then change again: It's a story familiar to developer managers everywhere. Projects are getting bigger and more complex. Developers are striving to reduce time-to-market, and squeeze releases out ever faster to keep pace with the market's demands for new functionality.

What's more, customers aren't prepared to wait until Q2 next year for a major release, to get the next batch of features built into the solution. They'd like them now, thanks very much — while reserving the right to change the feature set at any time. In these accelerated circumstances, speed of development is not just about the coding; there's an ever-growing emphasis on how you manage your teams and the overall development process.

Keeping up with these external and internal demands means setting clear ground rules, both for individual developers and teams. For example, what should happen when a developer has finished their part of the coding? How often should new code be shared within the team? Who should review it, and when? Should the build happen every time code is checked in?

And from the manager's viewpoint, the issues go still further. There's the need to ensure the code is reproducible. To select and manage the features that will make it into the final release of the code, set up the pre-release checklist, and perform any necessary security or compliance audits on the application.

So how do you keep serene, and keep pace? And ensure your team is working in harmony towards the same goals — and deadlines — without conflicts emerging, or developers getting sidetracked? The approach will depend on your development philosophy, and that of your team — such as whether you embrace agile development or use more traditional methods. But there are techniques that can benefit any development team, and help them to be nimbler and more responsive to changing demands.

Here are my five suggestions on how developer managers (and their teams) can stay fully in control of fast-moving projects.

Give Developers Their Own (Work)space

A desirable feature is for developers to have their own sandbox, or workspace, so they can build and test changes before they are integrated. This speeds up development — and so much

Issue	Best practice
Accommodate process	Flexible CM tool
Consistency	Central repository
Transparent audit trail, reproducibility	Task-based development
Reporting	Integrated issue tracking
Developer quality	Private workspaces / sandboxes
Developer productivity	Concurrent development / private workspaces
Feature scheduling/isolation	Streams, parallel development
Code re-use/sharing	Components
Distributed development	Client connection over LAN/WAN, caching server

A matrix of development issues versus best practice, to enable rapid response to changes.

the better if changes can be highlighted and reconciled automatically, enabling several developers to work on the same file concurrently. This approach enhances quality by allowing developers to build and test as they go along, prevents errors caused by incomplete submits, and increases productivity by allowing developers to work concurrently.

> Even if you're not a fan of parallel development, you can still use its techniques to your benefit

The right tool should also group individual file changes into a "changeset", making it easier to see and understand what has changed, and why, compared with the traditional approach of separate file changes. If changesets can be linked to the original change request to enhance transparency, even better.

This method gives the flexibility for developers to handle multiple tasks. For example, if an urgent bug fix comes in, developers can park their current work in a changeset, deal with the urgent fix, and then return to exactly where they left off, with all their files and changes preserved so they can restart quickly.

Consistency Matters

As well as helping developers do their job more efficiently, the CM tool should also aid developer managers in ensuring processes are consistent across his teams — giving the manager a top-level view of the status of current versions, who is working on what at a given time, and so on.

A key point is to have a configuration management (CM) solution that helps development teams work the way they want to — and doesn't force them to change the way they work. Can the solution integrate with the build tools your team is using, to save time, give rapid feedback about successful or failing builds, and avoid the risk of errors creeping in from having to constantly switch from one application to another?

Track Packages, Not Individual Changes

In the same way that the task-based approach, described earlier, benefits developers by grouping related changes into changesets, it also benefits the manager by giving an automated, real-time audit trail of changes and the ability to retrieve a configuration when needed — even when no label was created.

The changesets show the evolution of every project over time, including who made the change, why it was made, and to which files. This helps to put all the changes made in a project into context, in turn making it easier to match them to change requests. If the changesets are fully self-contained, they simplify rollbacks and parallel development,

As well as giving you better control of large or distributed teams, you get better transparency and easier status reporting. It's also a boon when revisiting older projects.

Parallel Lines

A common opinion about parallel development is that it's complex, lacks transparency, and creates an administrative headache in trying to track all the branching and merging.

But it doesn't have to be like that. With the workspace- and stream-based approach I've described, it's possible to isolate teams to work on different streams of a project, so they can edit, build and test separately. Then features can be merged back into the main code stream when ready. This also lets you decide what features you include in, or omit from, a release.

The key is to be able to keep different code streams in sync with each other so, for example, accepted changes could be shared automatically between streams. But done correctly, this means a broken build in one area doesn't stop the entire team from working. What's more, any bugs or unfinished changes stay contained in a stream, and don't spread to your whole project. So even if you're not a fan of parallel development, you can still use its techniques to your benefit.

Centralize and Share Code

Key to all of the suggestions I've made here is a central repository for all code in a project. As well as ensuring every developer is working on the same version, it also enables two short-cuts that can really save you time: the reuse of code and sharing components between projects.

The chances are that your team has worked on a project with elements that can be quickly adapted for a new project. If you're able to take a folder or component from one stream and share it with another, while automatically managing any changes or updates that are needed while the component is being worked on, then your team can capitalize on work it has already done, and boost its efficiency.

In conclusion, change is inevitable. But with the right processes and tools in place, you'll be better placed to take change in your stride, without it throwing you off course.

— Mike Shepherd is a configuration management specialist for PureCM Software Configuration Management.

Return to Table of Contents