United Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development August, 2009

Database Development

By Jonathan Erickson,
Editor In Chief

From e-science visualizations to Wall Street what-ifs, you know there's a problem when the talk turns to exabytes. But that problem isn't so much about too much data as it is about making sense of the data at hand. In other words, it's a question of data management—the architectures, policies, practices, and procedures you have in place for managing and enhancing a company's data assets.

What really stands out are the vendors that are providing tools to manage and analyze what's referred to as "big data." There are the usual suspects: Oracle, IBM, Google, Amazon.com, and FairCom. And then there are upstarts, such as Cloudera and Aster Data Systems, that are leveraging open source software such as MapReduce and Hadoop to build new businesses around big data.

Many of the technologies available to manage big data aren't new. In one form or another, column-oriented databases, data parallelism, solid-state drives, declarative programming languages, and cloud computing have been around for years. What's new is the emergence of "fringe databases," or database management systems that are appearing where you least expect sophisticated data management. For example, medical and consumer devices that once got by with flat files now require powerful database engines to manage the sheer volume of data being collected.

None of this comes without a price. What with big data on the rise, transaction throughput and concurrency requirements escalating, and data becoming more distributed, application complexity is increasing. To make it easier to manipulate data, it may have to be partitioned across multiple files or replicated and synchronized across multiple sites. And, of course, software developers are looking at complex data schema paradigms to accommodate their needs while still maintaining traditional relational access.

Hey, no one said it was going to be easy.

Return to Table of Contents

Bokode: A New Kind of Barcode

Tiny labels could pack lots of information, enable new uses

The ubiquitous barcodes found on product packaging provide information to the scanner at the checkout counter, but that's about all they do. Now, researchers at the MIT's Media Lab have come up with a new kind of very tiny barcode that could provide a variety of useful information to shoppers as they scan the shelves — and could even lead to new devices for classroom presentations, business meetings, video games, or motion-capture systems.

The system, called Bokode (http://web.media.mit.edu/~ankit/bokode/), is based on a new way of encoding visual information, explains MIT's Ramesh Raskar, who leads the lab's Camera Culture group (http://cameraculture.media.mit.edu/). Until now, there have been three approaches to communicating data optically:

- Through ordinary imaging (using two-dimensional space)
- Through temporal variations such as a flashing light or moving image (using the time dimension)
- Through variations in the wavelength of light (used in fiberoptic systems to provide multiple channels of information simultaneously through a single fiber).

But the new system uses a whole new approach, encoding data in the angular dimension: Rays of light coming from the new tags vary in brightness depending on the angle at which they emerge. "Almost no one seems to have used" this method of encoding information, Raskar says. "There have been three ways to encode information optically, and now we have a new one."

The new concept is described in the paper "Bokode: Imperceptible Visual Tags for Camera-based Interaction from a Distance", written by Ankit Mohan, Raskar, Grace Woo, Shinsaku Hiura, and Quinn Smithwick.

The tiny labels are just 3 millimeters across — about the size of the @ symbol on a typical computer keyboard. Yet they can contain far more information than an ordinary barcode: thousands of bits. Currently they require a lens and a built-in LED light source, but future versions could be made reflective, similar to the holographic images now frequently found on credit cards, which would be much cheaper and more unobtrusive.

"We're trying to make it nearly invisible, but at the same time easy to read with a standard camera, even a mobile phone camera," Mohan says.

One of the advantages of the new labels is that unlike today's barcodes, they can be "read" from a distance — up to a few meters away. In addition, unlike the laser scanners required to read today's labels, these can be read using any standard digital camera, such as those now built in to about a billion cellphones around the world.

The name Bokode comes from the Japanese photography term *bokeh*, which refers to the round blob produced in an out-of-focus image of a light source. The Bokode system uses an out-of-focus camera — which allows the angle-encoded information to emerge from the resulting blurred spot — to record the encoded information from the tiny tag. But in addition to being readable by any ordinary camera (with the

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
SERVICES MARKETING COORDINATOR
Laura Robison

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
Jill Thiry Publishing Director
Fritz Nelson Executive Producer, TechWeb TV

techweb™

focus set to infinity), it can also be read directly by eye, simply by getting very close — less than an inch away — to the tag.

As a replacement for conventional barcodes, the Bokode system could have several advantages, Mohan says. It could provide far more information (such as the complete nutrition label from a food product), be readable from a distance by a shopper scanning the supermarket shelves, and allow easy product comparisons because several items near each other on the shelves could all be scanned at once.

In addition to conventional barcode applications, the team envisions some new kinds of uses for the new tags. For example, the tag could be in a tiny keychain-like device held by the user, scanned by a camera in the front of a room, to allow multiple people to interact with a displayed image in a classroom or a business presentation. The camera could tell the identity of each person pointing their device at the screen, as well as exactly where they each were pointing. This could allow everyone in the room to respond simultaneously to a quiz, and the teacher to know instantly how many people, and which ones, got it right — and thus know whether the group was getting the point of the lesson.

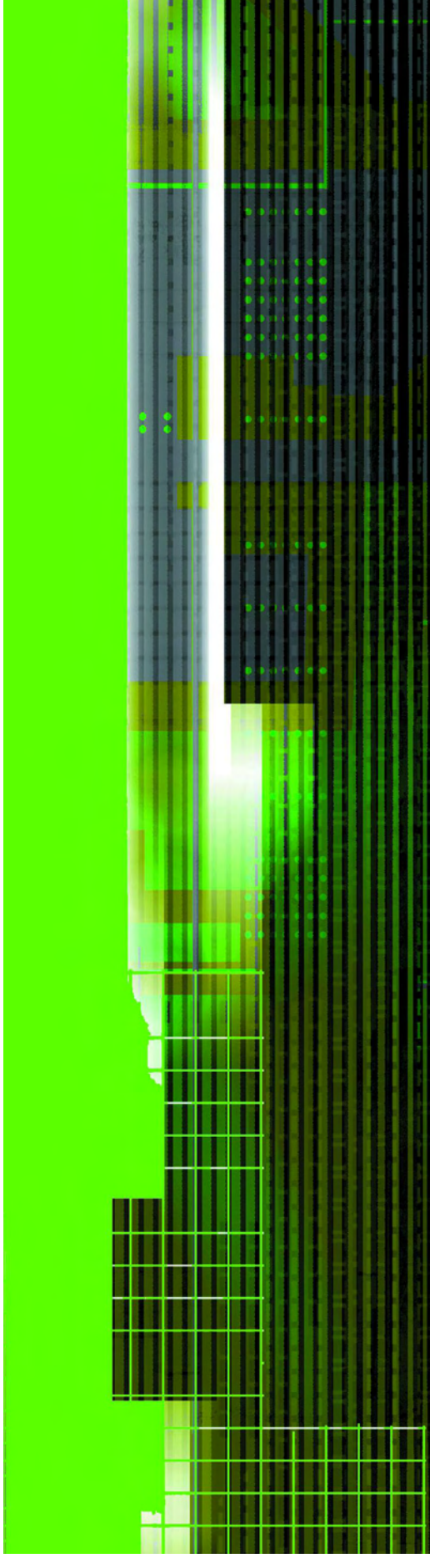
The devices could also be used for the motion-capture systems used to create videogames or computer-generated movie scenes. Typically, video cameras record a person or object's motions using colored dots or balls attached to various parts of the person's body. The Bokode system would allow the camera to record very precisely not just the position but the angle of each tag — with an accuracy of a tenth of a degree. This is far more accurate than any present motion-capture system.

Bokode "could enable a whole new range of applications," Raskar says. In the future, they could be used in situations such as museum exhibit labels, where the tiny codes would be unobtrusive and not detract from the art or other exhibit, but could send a whole host of background information to viewers through the use of their cellphone cameras. Or a restaurant could make its menu available to a passer-by on the sidewalk.

Bokode could also replace RFID systems in some near-field communication applications

It could also replace RFID systems in some near-field communication applications, Mohan suggested. For example, while RFIDs, now used in some ID cards, can provide a great deal of information, that information can be read from a distance, even when the card is inside a wallet. That makes them inappropriate for credit cards, for example, because the information could be retrieved by an unauthorized observer. But Bokode could encode just as much information, but require an open line-of-sight to the card to be read, increasing security.

The prototype devices produced at the Media Lab currently cost about $5 each, most of that cost due to use of an off-the-shelf convex glass lens, but Raskar says that price could easily drop to 5 cents once they are produced even in volumes of a few hundred units.



Return to Table of Contents

3 Steps To Managing Data In the Cloud

Matching the right cloud platform with the right database is critical

by Ken North

The emergence of cloud computing raises a host of questions about the best database technology to use with this new model for on-demand computing. Ultimately, the cloud approach a company chooses determines the data management options that are available to it.

When evaluating the suitability of a database manager for cloud computing, there are three basic steps:

- Consider the class of applications that will be served: data asset protection, business intelligence, e-commerce, etc.
- Determine the suitability of these apps for public or private clouds.
- Factor in ease of development.

The database manager you choose should be a function of the mission and the applications it supports, and not based on budgets and whether it will run in the enterprise as a private cloud or as a public cloud from a service provider. For instance, some companies turn to a cloud provider to back up mission-critical databases or as a disaster recovery option. Database-intensive apps such as business intelligence can be deployed in the cloud by having a SaaS provider host the data and the app, an infrastructure provider host a cloud-based app, or a combination of these approaches. And popular solutions for processing very large datasets, such as Hadoop MapReduce, can run in both the private and public cloud.

Databases, data stores, and data access software should be evaluated for suitability for both public and private clouds. Public cloud security isn't adequate for some types of applications. For example, Amazon Dynamo was built to operate in a trusted environment, without authentication and authorization requirements. At a minimum, database communications and backups to the cloud need to be encrypted.

Security in cloud environments varies based on whether you use SaaS, a platform provider, or an infrastructure provider. SaaS providers bundle tools, APIs, and services, so you don't have to worry about choosing the optimal data store and security model. But if you create a private cloud or use an infrastructure provider, you'll have to select a data management tool that's consistent with your app's security needs. Your database decision also will hinge on whether the environment supports a multitenant or multi-instance model. Salesforce.com hosts apps on Oracle databases using multitenancy. Amazon EC2 supports multi-instance security. If you fire up an Amazon Machine Image running Oracle, DB2, or Microsoft SQL Server, you have a unique instance that doesn't serve other tenants. You have to authorize database users, define roles, and grant user privileges when using the infrastructure-as-a-service model.

Developers' Choices

Database app development options for public cloud computing can be limited by the providers. SaaS offerings such as Google App Engine and Force.com provide specific development platforms with predefined APIs and data stores. Private cloud and infrastructure providers including GoGrid and Amazon EC2 let users match the software, database environment, and APIs to their needs. Besides cloud storage APIs, developers can program to various APIs for data stores and standard ones for SQL/XML databases. Programmers can work with SQL APIs and APIs for cloud services. For Amazon, that involves using Web Services Description

Language and invoking specific web services. For projects that use the cloud to power Web 2.0 apps, developers can use JavaScript Object Notation and the Atom Publishing protocol.

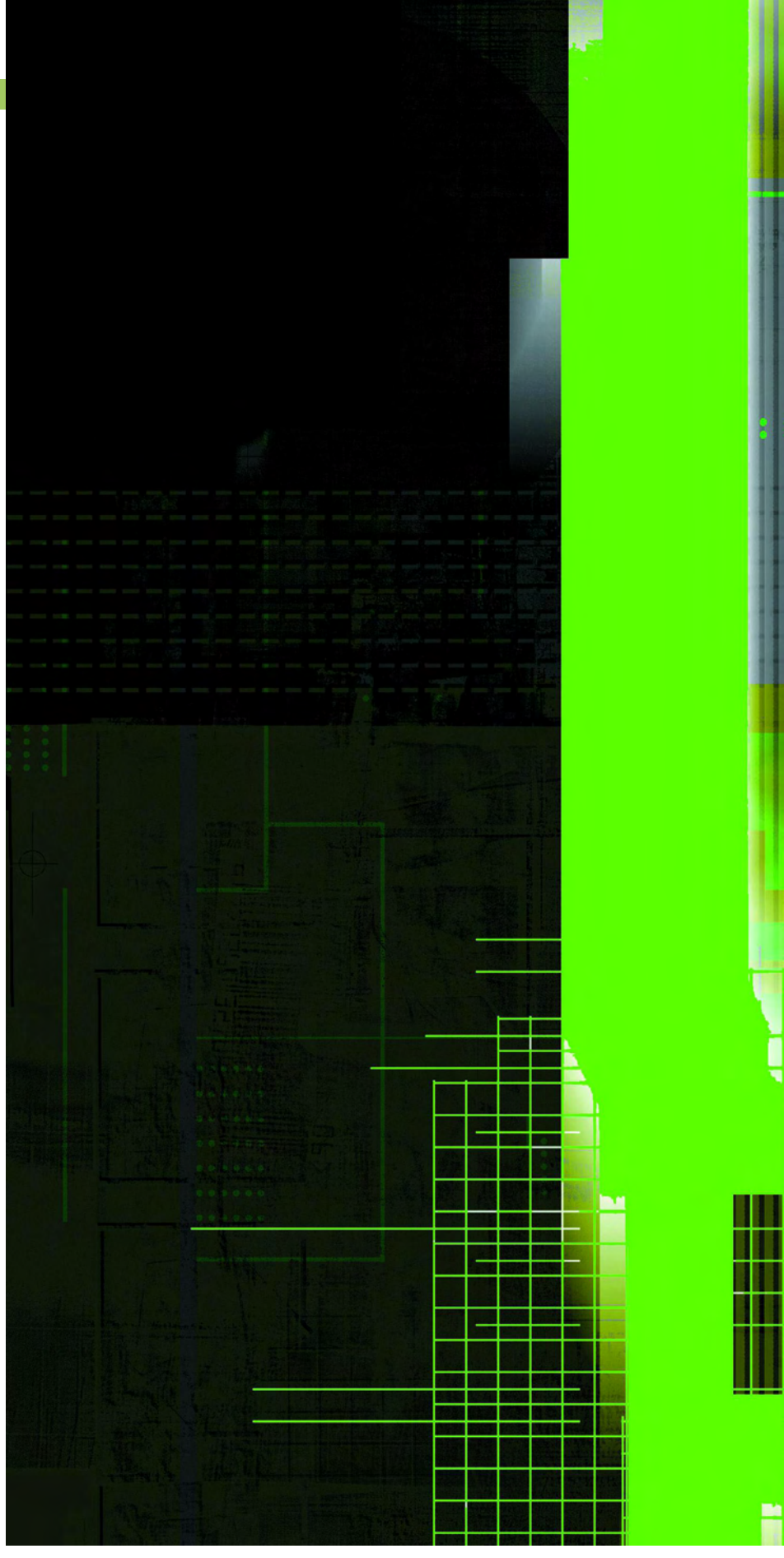
For new applications hosted in the cloud, developers look primarily to classes of data stores such as SQL/XML databases, column data stores, distributed hash tables, and tuple spaces variants, such as in-memory databases, entity-attribute-value stores, and other non-SQL databases. Choosing the right data store depends on the scalability, load balancing, consistency, data integrity, transaction support, and security requirements. Some newer data stores have taken a minimalist approach, avoiding joins and not implementing schemas or strong typing; instead, they store data as strings or blobs. Scalability is important for very large datasets and has contributed to the recent enthusiasm for the distributed hash table and distributed key-value stores.

One interesting approach is the ability to configure fault-tolerant systems and hot backups for disaster recovery. A private cloud can be configured and operated with fairly seamless failover to Amazon EC2, for example. You'll have to replicate data in the private and public cloud, implementing the Amazon APIs and availability zones, as well as IP assignment and load balancing for the private cloud. You'll also have to use server configurations compatible with Amazon instances to avoid breaking applications and services because of changes in endianness, the Java heap size, and other dissimilarities.

In short, the cloud is an effective elastic computing and data storage engine, but matching the right platform with the right database is critical. Doing this correctly requires evaluating the job and its security needs, as well as assessing how easy it is to design and implement the software. Carefully weighing these factors will lead you to the right conclusion.

— Ken North is an author, consultant, and analyst. He chaired the XML DevCon 200x conference series, Nextware, LinkedData Planet, and DataServices World 200x.

Return to Table of Contents



Databases in the Cloud: Elysian Fields or Briar Patch?

For handling distributed data in the cloud, the cornucopia of products includes everything from lightweight key-value stores to industrial-strength databases

by Ken North

Cloud computing is the latest sea change affecting how we develop and deploy services and applications and fulfill the need for persistent information and database solutions. Database technology evolves even as new computing models emerge, inevitably raising questions about selecting the right database technology to match the new requirements.

The cloud is an elastic computing and data storage engine, a virtual network of servers, storage devices, and other computing resources. It's a major milestone in on-demand or utility computing, the evolutionary progeny of computer timesharing, high-performance networks, and grid computing. The computer timesharing industry that emerged four decades ago pioneered the model for on-demand computing and pay-per-use resource sharing of storage and applications. More recently Ian Foster and Carl Kesselman advanced the concept of the grid to make large-scale computing networks accessible via a service model. Like computer timesharing and the grid, cloud computing often requires persistent storage so open source projects and commercial companies have responded with data store and database solutions.

Public clouds include commercial enterprises that can host applications and databases, offering Software as a Service (Saas), Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Database as a Service (DaaS). Infrastructure providers include Amazon Elastic Compute Cloud (EC2), GoGrid, Rackspace Mosso, and Joyent, whereas Microsoft Azure, Google AppEngine, Force.com, Zoho, and Facebook are platform providers. There are also providers targeting specific classes of cloud users, such as HP CloudPrint and IBM LotusLive for collaboration services and social networking for businesses. Other SaaS providers include Birst and SAS for on-demand business intelligence (BI), Salesforce.com and Zoho for customer relationship management (CRM), Epicor, NetSuite, SAP Business ByDesign, and Workday for enterprise resource planning (ERP) suites. The DaaS providers include EnterpriseDB, FathomDB, Longjump, and TrackVia.

Private clouds, like server consolidation, clusters, and virtualization, are another evolutionary step in data center and grid technology. Gartner Research predicted government will have the largest private clouds but any organization having thousands of servers and massive storage requirements is a likely candidate. Security and reliability are the appeal of private clouds for large enterprises that can afford the infrastructure. Public cloud computing does not provide the 99.99% uptime that enterprise data center managers desire for service-level agreements. The fact that a private cloud sits behind a firewall mitigates the risk from exposing data to the cloud. The private cloud also alleviates concerns about data protection in multitenancy cloud environments. One issue in the private versus public cloud debate is the diversity of APIs used to

invoke cloud services. This has caused interest in creating a standard, but the Eucalyptus initiative took a different approach. Assuming the Amazon APIs to be a de facto standard, it developed private cloud software that's largely compatible with Amazon EC2 APIs.

When evaluating the suitability of a database solution for cloud computing, there are multiple considerations.

First, you must consider the class of applications that will be served: business intelligence (BI), e-commerce transactions, knowledge bases, collaboration, and so on. Second, you must determine suitability for public and/or private clouds. Third, you must consider ease of development.

And, of course, budget is not to be overlooked.

Mission: What Will Run In the Cloud?

Selecting a database manager should be a function of the mission and applications it must support, not just budget and whether it will run in the enterprise or a private or public cloud. Some organizations use a cloud provider as a backup for mission-critical applications or databases, not as the primary option for deploying applications or services. Oracle database users can run backup software that uses Amazon Simplified Storage System (S3) for Oracle database backups. For an even bigger safety net, organizations can look to cloud computing as a disaster recovery option.

The New York Times project that created the TimesMachine, a web-accessible digital archive, is a prime example of a one-off cloud project requiring massively scalable computing. But factors besides on-demand elasticity come into play when the goal is hosting applications in the cloud on a long-term basis, particularly database applications.

Cloud users are often looking to deploy applications and databases with a highly scalable, on-demand architecture, often on a pay-per-use basis. Common scenarios for using the cloud include startups and project-based, ad hoc efforts that want to ramp up quickly with minimal investment in infrastructure. But Amazon's public cloud has also been used to support e-business websites, such as NetFlix.com, eHarmony.com, and Target.com. E-mail is a backbone of modern business and companies, such as the Boston Celtics, have gone to a cloud computing model for e-mail and collaboration software. Companies can also opt to use a cloud to host ERP or CRM suites that operate with SQL databases, such as open source ERP suites (Compiere, Openbravo, SugarCRM) and BI solutions (Jasper, Pentaho). Because data warehouses use source data from operational systems, organizations using the cloud to host operational databases are likely to do the same for data warehouses and business intelligence.

On a pay-as-you-go basis, the cloud handles provisioning on demand. Machine images, IP addresses, and disk arrays are not permanently assigned, but databases on a public cloud can be assigned to persistent storage. This saves having to bulk load a database each time you fire up machine instances and run your application. But it also puts a premium on database security and the cloud provider having a robust security model for multitenancy storage.

The cloud is particularly well-suited for processing large datasets and compute-intensive applications that benefit from parallel processing, such as rendering video and data analysis. The early Amazon EC2 users have included biomedical researchers, pharmaceutical, bioengineering, and banking institutions. They were early adopters of grid computing for purposes such as financial modeling, drug discovery, and other research. Medical research often requires massive simulations of genetic sequencing and molecular interactions. This has been done by grids, often using Basic Local Alignment Search Tool (BLAST) programs, and more recently by clouds. Researchers have also used MapReduce software in the cloud for genetic sequence analysis. Eli Lilly uses Amazon EC2 for processing bioinformatics sequence information.

Cloud computing is also used for other purposes, such as integrating SaaS and enterprise systems. Players in the on-demand integration space include Boomi, Cast Iron Systems, Hubspan, Informatica, Jitterbit, and Pervasive Software. Business intelligence (BI) activity, such as analytics, data warehousing, and data mining, requires horsepower and a capital outlay that might be prohibitive for small and medium businesses. Cloud computing offers an attractive pay-per-use alternative and there appears to be a large potential BI-on-demand market.

The marriage of cloud computing and business intelligence can be accomplished by several means. One option is to have data and applications hosted by a SaaS provider. Another is to create cloud-based applications hosted by an infrastructure provider. A third alternative is to do both and use data replication or a data integration suite.

PaaS, Public, Private Clouds

The Platform-as-a-Service (PaaS) solution bundles developer tools and a data store, but users who opt to use an infrastructure provider or build a private cloud have to match the data store or database to their application requirements and budget. There are open source and commercial products that have a wide range of capabilities, from scalable simple data stores to robust platforms for complex query and transaction processing.

Databases, data stores, and data access software for cloud computing must be evaluated for suitability for both public and private clouds and for the class of applications supported. For example, Amazon Dynamo was built to operate in a trusted environment, without authentication and authorization requirements. Whether the environment supports multitenant or multi-instance applications also influences the database decision.

Databases and Data Stores

Data management options for the cloud include single format data stores, document databases, column data stores, semantic data stores, federated databases, and object-relational databases. The latter group includes "Swiss Army Knife" servers from IBM, Microsoft, OpenLink, and

Oracle that process SQL tables, XML documents, RDF triples, and user-defined types.

Building a petabyte size web search index is a very different problem from processing an order or mapping wireless networks. The requirements of the application and data store for those tasks are quite different. For new applications hosted in the cloud, developers will look primarily to several classes of data store:

- SQL/XML (object-relational) databases
- Column data stores
- Distributed hash table (DHT), simple key-value stores
- Tuple spaces variants, in-memory databases, entity-attribute-value stores, and other non-SQL databases having features such as filtering, sorting, range queries, and transactions.

Because this cornucopia of data stores has diverse capabilities, it's important to understand application requirements for scalability, load balancing, consistency, data integrity, transaction support, and security. Some newer data stores are an exercise in minimalism. They avoid joins and don't implement schemas or strong typing, instead storing data as strings or blobs. Scalability with very large dataset operations is a requirement for cloud computing, which has contributed to the recent enthusiasm for the DHT and distributed key-value stores.

Associative arrays, dictionaries, hash tables, rings, and tuple spaces have been around for years, as have entity-attribute-value (EAV) stores, database partitions and federated databases. But cloud computing puts an emphasis on scalability and load balancing by distributing data across multiple servers. The need for low-latency data stores has created an Internet buzz about key-value stores, distributed hash tables (DHT), entity-attribute-value stores, and data distribution by sharding.

Tuple spaces are a solution for distributed shared memory that originated with the Linda effort at Yale that spawned more than 20 implementations, including Object Spaces, JavaSpaces, GigaSpaces, LinuxTuples, IBM TSpaces, and PyLinda. One can find GigaSpaces eXtreme Application Platform as a pay-per-use service on Amazon EC2. It includes a local and distributed Jini transaction manager, Java Transaction API (JTA), JDBC support, with b-tree and hash-based indexing capabilities. Amazon SimpleDB also provides standard tuple spaces interfaces, but adds secondary indexing and support for additional query operators.

For large datasets and databases, partitioning data has been a facilitator of parallel query processing and load balancing. Horizontal partitioning, referred to as sharding, has caught the attention of developers looking to build multiterabyte cloud databases because of its success at Amazon, Digg, eBay, Facebook, Flickr, Friendster, Skype, and YouTube.

SQLAlchemy and Hibernate Shards, object-relational mappers for Python and Java, respectively, provide sharding that's useful for cloud database design. Google developed Hibernate Shards for data clusters before donating it to the Hibernate project. You can do manual sharding for a platform such as Google AppEngine, use SQLAlchemy or Hibernate Shards for Python or Java development, or use a cloud data store such as MongoDB that provides administrative commands for creating shards.

Distributed Hash Table, Key-Value Data Stores

Distributed hash tables and key-value stores are tools for building scalable, load balanced applications, not for enforcing rigid data integrity, consistency, and Atomic Consistent Isolated Durable (ACID) properties for transactions. They have limited applicability for applications doing ad hoc query and complex analytics processing. Products in this group include memcached, MemcacheDB, Project Voldemort, Scalaris, and Tokyo Cabinet. Memcached is ubiquitous and a popular solution for caching database-powered web sites. It's a big associative array that's accessed with a *get* or *put* function, using the key that's a unique identifier for data. It's particularly useful for caching information produced by expensive SQL queries, such as counts and aggregate values. MemcacheDB is a distributed key-value data store that conforms to the memcached protocol but uses Berkeley DB for data persistence.

Scalaris is a distributed key-value store, implemented in Erlang, which has a non-blocking commit protocol for transactions. Using the web interface, you can read or write a key-value pair, with each operation being an atomic transaction. Using Java, you can execute more complex transactions. Scalaris has strong consistency and supports symmetric replication, but does not have persistent storage.

The open source Tokyo Cabinet database library is causing a buzz in online discussions about key-value stores. It's blazingly fast, capable of storing one million records in 0.7 seconds using the hash table engine and 1.6 seconds using the b-tree engine. The data model is one value per key and it supports LZW compression. When keys are ordered, it can do prefix and range matching. For handling transactions, it features write ahead logging and shadow paging. Tokyo Tyrant is a database server version of Tokyo Cabinet that's been used to cache large SQL databases for high-volume applications.

Some products of this group support queries over ranges of keys, but ad hoc query operations and aggregate operations (sum, average, grouping) require programming because they are not built-in.

Hadoop MapReduce

Hadoop MapReduce would be a nominee for the Academy Award for parallel processing of very large datasets, if one existed. It's fault-tolerant and has developed a strong following in the grid and cloud computing communities, including developers at Google, Yahoo, Microsoft, and Facebook. Open source Hadoop is available from Apache, a commercial version is available from CloudEra, and Amazon offers an Elastic MapReduce service based on Hadoop.

MapReduce operates over the Hadoop Distributed File System (HDFS), with file splits and data stored as key value pairs. The HDFS enables partitioning data for multiple machines to do parallel processing of batches and reduce processing time.

MapReduce is suitable for processing very large datasets for purposes such as building search index engines or data mining, but not for online applications requiring sub-second response times. Frameworks built on top of Hadoop, such as Hive and Pig, are useful for extracting information from databases for Hadoop processing. The eHarmony.com site is an example of the marriage of an Oracle database and Amazon MapReduce, using the latter for analytics involving millions of users.

Entity-Attribute-Value Datastores

EAV stores are derived from data management technology that predates the relational model for data. They do not have the full feature set of an SQL DBMS, such as a rich query model based on a nonprocedural, declarative query language. But they are more than a simple key-value data store. EAV data stores from major cloud computing providers include Amazon SimpleDB, Google AppEngine data store, and Microsoft SQL Data Services. And one type, the RDF datastore used for knowledge bases and ontology projects, has been deployed in the cloud.

Google Bigtable uses a distributed file system and it can store very large datasets (petabyte size) on thousands of servers. It's the underlying technology for the Google AppEngine data store. Google uses it, in combination with MapReduce, for indexing the Web and for applications such as Google Earth. Bigtable is a solution for projects that require analyzing a large collection, for example the one billion web pages and 4.78 billion URLs in the ClueWeb09 dataset from Carnegie Mellon University. For those seeking an open source alternative to Bigtable for use with Hadoop, Hypertable and HBase have developed a following. Hypertable runs on top of a distributed filesystem, such as HDFS. HBase data is organized by table, row, and multivalued columns and there's an integrator-style interface for scanning a range of rows. Hypertable is implemented in C++, whereas HBase is implemented in Java.

The Google AppEngine includes a schemaless data store that's optimized for reading, supports atomic transactions and consistency, and stores entities with properties. It permits filtering and sorting on keys and properties. It has 21 built-in data types, including list, blob, postal address, and geographical point. Applications can define entity groupings as the basis for performing transactional updates and use GQL, a SQL-like query language. Access to the Google AppEngine data store is programmable using Python interfaces for queries over objects known as entities. The data store is also programmable using Java Data Objects (JDO) and the Java Persistence API. Although AppEngine bundles a data store, the AppScale project provides software for operating with data stores such as HBase, Hypertable, MongoDB, and MySQL.

Amazon Platform

Amazon SimpleDB is a schemaless, Erlang-based, eventually consistent data store suited for high-availability applications. The data model provides domains of large collections of items, which are hash tables containing attributes that are key-value pairs. Attributes can have multiple values and there are no joins. The query language provides queries that can return an *itemName*, all attributes, the attribute count, or an attribute list. Data is stored in a single format (untyped strings), without applying constraints, so all predicate comparisons are lexicographical. Therefore, for accurate query results you must store data in an ordered format; for example, padding numbers with leading zeroes and using dates in ISO 8601:2004 format.

Azure Services Platform

Microsoft Azure, like Google AppEngine and Force.com, offers a platform for cloud computing that includes a data store and other features for application development. Microsoft .NET Services provide a service bus and authentication and Live Services are application building blocks. Microsoft also offers SharePoint Services and Dynamics CRM Services in the Azure cloud. Like Amazon S3 and EC2, communication using the Azure Services Platform is based on the web services model, with Microsoft supporting SOAP and REST. Microsoft Azure bundles SQL Data Services (SDS) and exposes Azure Table Storage via ADO.NET Data Services. The database Azure currently offers is a single instance of SQL Server that's limited to 10 gigabytes of storage. For a larger requirement it's necessary to partition data to scale horizontally.

For those with a history of using industrial-strength databases, a big adjustment to the new EAV stores is lack of strong typing. SimpleDB uses string values to store everything, so comparisons and sorting require that you pad numbers with leading zeros. Microsoft SQL Data Services provides Base64, Boolean, datetime decimal, and string. With more than 20 types, Google AppEngine has more built-in types than SimpleDB or SQL Data Services.

RDF and Semantic Data Stores

Social networking and e-commerce have shown us there are classes of web applications that must operate with massive data stores and support a user base measured in millions. Cloud computing is often touted as a vehicle for scaling out that type of site and powering Web 3.0 applications. Tim Berners-Lee has said a web of linked data will evolve from the web of linked documents. This has produced a surge of interest in data stores that can handle very large knowledge bases and datasets encoded to impart semantics using the W3C Resource Description Format (RDF) and in the W3C SPARQL query language.

Interest in RDF, micro formats, and linked data has raised awareness of the capabilities and capacity of RDF data stores. Because there are a number of RDF data stores, the benchmark wars are reminiscent of the Transaction Processing Council (TPC) benchmark competition among SQL vendors. RDF data is stored as subject-predicate-object triples. The leading RDF data stores often store additional information for versioning and temporal queries, but they are capable of storing and querying over billions of triples. A W3C wiki identifies more than a dozen triple stores, about half citing deploy-

ments or benchmarks with one billion triples or more.

Sesame, Jena, and Mulgara are popular open source solutions. OpenLink Virtuoso is a universal server that in a recent benchmark loaded 110,500 triples per second. The Virtuoso Universal Server (Cloud Edition) is a prepackaged AMI for EC2. In addition to SQL and XML databases, it provides online backup to Amazon S3 buckets and installable RDFizer cartridges. Franz AllegroGraph RDFStore offers a vehicle for building RDF-based federated knowledge stores in the cloud. It supports SPARQL queries, Prolog, and RDFS++ reasoning. On Amazon EC2, it stored and indexed a 10-billion triple dataset in 6.19 hours using 10 large EC2instances. The SQL/XML products can store RDF triples, including Oracle11g, IBM Boca for DB2. On the patent front, Microsoft has been active with applications for methods to store RDF triplets and convert SPARQL queries to SQL.

Document Stores, Column Stores

Storing data by column, rather than the row-orientation of the traditional SQL platform, does not subvert the relational model. But when combined with data compression and a shared-nothing, massively parallel processing (MPP) architecture, it can sustain high-performance applications doing analytics and business intelligence processing. By using a Sybase IQ or Vertica column store with a cloud computing service, organizations can roll their own scalable BI solutions without a heavy capital outlay for server hardware. Sybase IQ processes complex analytics queries, accelerates report processing and includes a word index for string processing, such as SQL LIKE queries. It provides connectivity via standard data access APIs and its Rcube schemas provide a performance advantage over the star schema typically used for relational data warehouses and data marts. Vertica Analytic Database is a solution from a company co-founded by Michael Stonebreaker. Vertica supports a grid architecture, terabyte-sized databases, and standards-based connectivity. It makes pay-as-you-go analytics available to Amazon EC2 users, with a large AMI instance, drivers for ODBC, JDBC, Python, and Ruby, and a database size of 1 terabyte per node as you scaleout to multiple nodes.

Apache CouchDB is a schema-less, fault-tolerant data store that appeals to developers building HTTP applications for which a document paradigm is useful. It supports the JavaScript Object Notation (JSON) and AtomPub data formats and it provides a REST-style API for reading and updating named documents. To ensure data consistency, it has ACID properties and does not overwrite committed data. It uses a document ID and sequence number to write a b-tree index, with sequence number providing the ability to generate a document's version history. CouchDB provides a peer-based distributed database solution that supports bidirectional replication.

SQL/XML Databases

The SQL database has survived every paradigm shift critics said would be the death of SQL, including object-oriented programming (OOP), online analytical processing (OLAP), Internet computing, and the World Wide Web. Some have suggested SQL platforms are not sufficiently scalable for large workloads or data volumes, but there's ample evidence to the contrary. The UPS shipping system central database processes 59 million transactions per hour. It has a table that contains more than 42 billion rows and has achieved a peak workload of more than 1 billion SQL statements per hour with IBM DB2. The data warehouse at eBay, running on a Teradata system, contains 5 petabytes of data. LGR Telecommunications derives information from call records to feed a 310 TB Oracle data warehouse. At a recent conference, Microsoft reported Hotmail has 300 million users and processes more than 2 billion non-spam messages per day with Microsoft SQL Server running on a 10,000 server farm.

The SQL/XML database platforms provide a rich query model, supporting SQL, XQuery, XPath expressions, and SPARQL queries. Typically, a key-value store requires logic in the application to perform record-oriented query processing. But instead of procedural programming, the SQL solution offers a declarative programming solution that relies on the query optimizer to generate the access path to the data. The SQL platforms offer mature administrative tools and standards-based connectivity, but the highest capacity SQL configurations have not yet been seen in the pay-per-use cloud. IBM DB2 gives you a hybrid storage engine that supports transaction processing, business intelligence, and XML document processing. It currently holds several TPC benchmark records, including one million TPC-C transactions per minute on an 8 processor/64 core cluster running Red Hat Linux Advanced Server. But the ready-to-run Amazon EC2 AMIs aren't configured for that type of workload. The AMI's bundles are for running IBM DB2 Express Edition or Workgroup Edition and Informix Dynamic Server Express Edition and Workgroup Edition. For heavier lifting, you'll need to move your own DB2 Enterprise Edition or Informix Dynamic Server (IDS) licenses to EC2. Besides DB2 and Informix Dynamic Server, there are prepackaged AMIs for IBM Lotus Web Content Management and Web Sphere sMash. For DB2 or IDS development, IBM provides Developer AMIs for EC2 that have no DB2 or IDS usage charge.

Oracle users can transfer licenses to EC2 for the Oracle 11g database,Fusion Middleware, and Enterprise Manager. The company also provides ready-to-run AMIs and the Oracle Secure Backup Cloud Module can create compressed and encrypted database backups using Amazon S3. The S3 backups easily integrate with Oracle Recovery Manager using its SBT interface. The Oracle EC2 AMIs are preconfigured to use Enterprise Linux. The selection includes Oracle Database 10g Express Edition, Oracle Database 11g Enterprise Edition, Oracle Database 11g SE, and WebLogic Server 10g. Oracle's licensing policy permits moving Fusion Middleware to EC2, including WebLogic Server, JRockit (Java VM), Coherence, and the Tuxedo transaction processing monitor.

Oracle Coherence is an in-memory, distributed data grid that stores key-value pairs. It provides linear scalability

(reportedly deployed ina 5000-node grid), replication, caching, and transparent fail over for distributed data. Coherence supports analysis and aggregation over the entire grid and it's available for C++, Java, and .NET development. Oracle Real Application Clusters are not currently available on a public cloud provider.

MySQL Enterprise is a platform suitable for cloud computing scenarios, such as scaling out with multiple servers and using master/slave replication. Some MySQL users have created a high-availability solution for Amazon EC2 by using a multi-instance master-master replication cluster. MySQL Enterprise subscribers can sign up for 24x7 support services for EC2, with different levels of support available from Sun. With the Platinum subscription, you get an enterprise dashboard, replication monitor, connectors, caching (memcached) and partitioning with MySQL Advanced. Continuent and Sun are working on making MySQL clustering technology available on cloud computing services such as GoGrid, Rackspace Mosso, and Amazon EC2.

EnterpriseDB Postgres Plus Cloud Edition is a version of PostgreSQL with enhancements such as GridSQL, replication, asynchronous pre-fetch for RAID, and a distributed memory cache. GridSQL uses a shared-nothing architecture to support parallel query processing for highly scalable environments such as grids and clouds. For its cloud edition, EnterpriseDB partnered with Elastra, which had a SaaS offering with PostgreSQL and MySQL on Amazon and a product for management of clustered data warehouses. Elastra used Amazon S3 as the persistence solution to the ephemeral disk storage problem when trying to manage databases using EC2 instances.

In-Memory Databases, Cache

For applications that require extremely high throughput, in-memory databases and caches can be deployed to deliver performance. One solution is to pair an in-memory database with a disk-resident SQL database, with the former acting as a cache for the latter. Times Ten and solidDB are robust in-memory products that were acquired by Oracle and IBM, respectively. Oracle Times Ten is an embeddable, in-memory database that supports ODBC and JDBC data access. It can provide real-time caching for and automatic synchronization with Oracle 11g databases. IBM solidDB maintains redundant copies of the database at all times and provides JDBC and ODBC query capabilities. It can scale using partitioning for instances, act as cache for SQL databases, and do periodic snapshots (checkpoints to disk). The GigaSpaces XAP builds on the tuple spaces shared memory model and offers a JDBC capability for SQL queries.

To improve responsiveness, high-volume web sites typically use cache to reduce the number of queries against SQL databases. Ehcache is a distributed Java cache used by LinkedIn. The memcached server provides a distributed object cache often used for MySQL applications. JBoss Cache has been integrated with GridGain in the Open Cloud Platform. GridDynamics demonstrated linear scalability with the GridGain platform from 2 to 512 nodes using Amazon EC2 to run Monte Carlo simulations.

Quetzall CloudCache is targeted at cloud applications hosted on EC2. CloudCache returns data in JSON or XML format and it can run in multiple EC2 regions. It offers a REST-style API and there are bindings for Ruby, Python, PHP, and Java development. Microsoft is currently previewing a distributed, in-memory cache code-named Velocity. It supports retrieving data by key or tag, optimistic and pessimistic concurrency, and it includes an ASP.NET session provider.

To improve responsiveness, high-volume web sites typically use cache to reduce the number of queries against SQL databases

Federated Data

The federated database provides a solution when data is distributed because volume, workload, or other considerations make it impractical to combine it into a single database. Open SkyQuery and Flickr have been showcases for federation.

SkyQuery runs distributed queries over federated astronomical data sources. Flickr uses sharding to support billions of queries per day over federated MySQL databases used for data management of 2 billion photos. That type of success and the scalability requirements of cloud computing have put new emphasis on federated data and sharding. Mergers and acquisitions also may force the creation of federated data stores to permit execution of business intelligence and other queries against disparate CRM databases.

IBM has been using GaianDB, based on Apache Derby, to test performance of a lightweight federated database engine. It distributed the database over 1000 nodes, which GaianDB was able to query in 1/8 of a second. Fetching a million rows took five seconds.

Platform And API Issues

Database options for public cloud computing can be limited by the choice of cloud provider. SaaS providers, such as Google AppEngine and Force.com, offer a specific platform for development, including predefined APIs and data stores. But private clouds and infrastructure providers, such as GoGrid, Joyent, and Amazon EC2,

enable the cloud user to match the software, database environment, and APIs to requirements.

Besides cloud storage APIs, developers can program to diverse APIs for data stores and standards-based APIs for SQL/XML databases. The programmer developing applications for the cloud can work with SQL APIs and APIs for cloud services. For Amazon, that involves using Web Services Description Language (WSDL) and invoking specific web services. For projects that use the cloud to power rich Internet applications (Web2.0), developers might be looking to use JavaScript Object Notation (JSON) and the Atom Publishing protocol (AtomPub). More than one guru considers AtomPub to be the de facto standard for accessing cloud-based data.

Ease of development is an important aspect for a cloud database solution, with application programming interfaces (API) being a major factor. Some data access programming for the cloud can be done with familiar APIs, such as Open Database Connectivity (ODBC), JDBC, Java Data Objects (JDO), and ADO.NET.

Security

For certain classes of applications, security is an obstacle to using public cloud services, but it's not insurmountable. Current thinking on the subject emphasizes encryption, authorization, authentication, digital certificates, roles, and policy-based security controls. Database backups to the cloud can be encrypted. Communications can use secure networking and encrypted data.

Java and .NET offer robust cryptographic solutions for applications and services accessing databases. Operating systems and robust SQL databases offer additional layers of security. SQL databases provide features such as row-level encryption and role-based assignment of privileges and access to data. But even with multilevel security, one serious threat to databases in the public cloud and the corporate data center is a breach of hypervisor security by an authorized employee. In order to ensure data security, Amazon EC2 provides for the definition of security groups. But you must use an Amazon API function to manually monitor the security group descriptions. And there is no logging function to monitor failed attempts at authentication.

There are differences in security depending on whether you use SaaS, a platform provider, or infrastructure provider. Because SaaS providers offer a bundle with tools, APIs and services, the SaaS user is not caught up in choosing the optimal data store and security model. However, those creating private clouds or using an infrastructure provider must select a data management solution that's consistent with the application's security requirement.

Salesforce.com hosts applications on Oracle databases using a multitenancy model. On the other hand, Amazon EC2 is an example of multi-instance security. If you fire up an AMI running Oracle, DB2 or Microsoft SQL Server, you have a unique instance that does not serve other tenants. The process of authorizing database users, defining roles, and granting privileges is your responsibility when using IaaS.

Fault-Tolerance And Cloud Fail Over

One of the exciting possibilities introduced by cloud service providers is being able to configure fault-tolerant, highly available systems and hot backups for disaster recovery. It's possible to configure and operate a private cloud for a fairly seamless fail over to Amazon EC2, for example. It would require replicating data in the private and public cloud, implementing the Amazon APIs and availability zones, IP assignment and load balancing for the private cloud, and using server configurations compatible with Amazon instances. The latter would be necessary to avoid breaking applications or services due to changes in endianness, the Java heap size, and other dissimilarities. A recent dialogue with IBM about the inverse scenario, deploying databases in a public cloud and moving them to a private cloud, revealed this would be more of a challenge.

Final Thoughts

The SQL database became predominant even though an earlier generation of databases delivered ACID properties and excellent performance on Create Replace Update Delete (CRUD) operations. They, like some of the software mentioned here, required a programmer to write code to navigate through data in order to perform queries, such as an aggregation query. But SQL platforms provided an ad hoc query solution that did not require procedural programming because it used a declarative query language and provided built-in aggregation functions. The logic that must be programmed in an application or service, versus built-in with the database engine, is an element of the total cost of ownership (TCO) of a data store solution.

The direction an organization takes on cloud computing, whether to go the private cloud route or use a public cloud, will determine what options are available for data management. For those who walk the PaaS path, the focus will be on the platform's capabilities, not the data store per se. Those who walk the private cloud or IaaS paths will have to choose a hardware and software configuration, including a data store that fits the business goals and requirements of applications running in the cloud. Many factors will influence the choice of one or more of a spectrum of cloud database solutions, ranging from simple data stores to platforms that support complex queries and transaction processing.

Not every project requires the full functionality of the SQL database managers so there's a definite need for lightweight, fast, scalable data stores.

—Ken North is an author, consultant and industry analyst. He teaches seminars and chaired the XML Devcon 200x conference series, Nextware, LinkedData Planet and DataServices World 200x conferences.

Return to Table of Contents

Fan: A Portable Language Is Bringing Actors to JavaScript

Fan is both an object-oriented and functional programming language

by Gastón Hillar

Fan (http://www.fandev.org/), another new programming language developed in the multicore era, has recently launched its 1.0.45 release (http://code.google.com/p/fan/downloads/list). It is a very active open source project with an interesting approach to many modern concurrent programming challenges.

I began writing about Fan 1.0.44 a week ago. Now, Fan has a new version, 1.0.45.

Most developers don't want to learn a new programming language. However, Fan is an attractive language for certain tasks because it is trying to solve modern problems related to concurrency and multicore programming.

Fan is both an object-oriented and functional programming language. This means that a developer can combine functional programming code with object-oriented code. However, at the same time, it has built-in immutability, message passing, and REST-oriented transactional memory. It uses Java-style syntax. Therefore, Java and C# developers won't have problems understanding Fan code.

Its portability makes Fan unique. So far, it can run on the JVM (Java Virtual Machine), the .NET CLR (Common Language Runtime), and JavaScript. Its JavaScript support is one of the most exciting features I've found for this language. There are many other languages and libraries offering actors and many different concurrency models for the JVM and the .NET CLR. However, Fan's support for JavaScript could revolution the scripting performance. Scripting could take advantage of multicore.

In fact, scripting should take advantage of multicore. Fan is evolving to offer JavaScript developers the possibility to tackle concurrency using actors and message-passing features. Besides, you can also run it on the JVM or on the .NET CLR. However, you can expect Fan to offer additional compilers to run on new platforms. Portability is very important for Fan.

Fan creators talk about the productivity of Ruby with the performance of Java. As Fan offers the possibility to tackle multicore, developers can transform a slow performance language into a fast one. Undoubtedly, JavaScript's support is a great opportunity for developers to create higher performance RIAs (Rich Internet Applications) based on this popular scripting language.

It's a bit difficult to find a definition for Fan, because it tries to offer many different features in one single and simple language. In a few lines, this list offers a summary of Fan's main features:

- Portability
- Object-oriented (with inheritance support)
- Immutability
- Closures support
- Dynamic programming
- Functional programming
- Serialization support
- Actor framework

The actor framework is really powerful. It supports the most important features required to create concurrent code without problems:

- Actor locals
- Actor pools (using a shared thread pool)
- Futures
- Timers
- Chaining
- Message passing
- Coalescing messages
- Flow control mechanisms

If you need functional programming, immutability, message passing, and actors, you should take a look at Fan. Likewise, if you are working with JavaScript, keep an eye on Fan. It can help developers to tackle multicore.

In the forthcoming months, expect to see new libraries, languages, compilers, and Domain-Specific Languages appearing to simplify parallel programming for many languages, virtual machines, and runtimes.

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

Return to Table of Contents

The C++0x "Remove Concepts" Decision

"Concepts" were to have been the central new feature in C++0x

By Bjarne Stroustrup

At the July 2009 meeting in Frankfurt, Germany, the C++ Standards Committee voted to remove "concepts" from C++0x. Although this was a big disappointment for those of us who have worked on concepts for years and are aware of their potential, the removal fortunately will not directly affect most C++ programmers. C++0x will still be a significantly more expressive and effective language for real-world software development than C++98. The committee acted with the intent to limit risk and preserve schedule. Maybe a significantly improved version of "concepts" will be available in five years. This article explains the reasons for the removal of "concepts," briefly outlines the controversy and fears that caused the committee to decide the way it did, gives references for people who would like to explore "concepts," and points out that (despite enthusiastic rumors to the contrary) "the sky is not falling" on C++.

No "Concepts" in C++0x

At the July 2009 Frankfurt meeting of the ISO C++ Standards Committee (WG21) (http://www.open-std.org/jtc1/sc22/wg21/), the "concepts" mechanism for specifying requirements for template arguments was "decoupled" (my less-diplomatic phrase was "yanked out"). That is, "concepts" will not be in C++0x or its standard library. That — in my opinion — is a major setback for C++, but not a disaster; and some alternatives were even worse.

I have worked on "concepts" for more than seven years and looked at the problems they aim to solve much longer than that. Many have worked on "concepts" for almost as long. For example, see (listed in chronological order):

- Bjarne Stroustrup and Gabriel Dos Reis: "Concepts — Design choices for template argument checking." October 2003. An early discussion of design criteria for "concepts" for C++.
- Bjarne Stroustrup: "Concept checking — A more abstract complement to type checking." October 2003. A discussion of models of "concept" checking.
- Bjarne Stroustrup and Gabriel Dos Reis: "A concept design (Rev. 1)." April 2005. An attempt to synthesize a "concept" design based on (among other sources) N1510, N1522, and N1536.
- Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi, and Andrew Lumsdaine: "Concepts for C++0x." N1758==05-0018. May 2005.
- Gabriel Dos Reis and Bjarne Stroustrup: "Specifying C++ Concepts." *POPL06*. January 2006.
- D. Gregor, B. Stroustrup: "Concepts." N2042==06-0012. June 2006. The basis for all further "concepts" work for C++0x.
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: "Concepts: Linguistic Support for Generic Programming in C++." *OOPSLA'06*, October 2006. An academic paper on the C++0x design and its experimental compiler "ConceptGCC."
- Pre-Frankfurt working paper (with "concepts" in the language and standard library): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2914.pdf. N2914=09-0104. June 2009.
- B. Stroustrup: "Simplifying the use of concepts." N2906=09-0096. June 2009.

It need not be emphasized that I and others are quite disappointed. The fact that some

alternatives are worse is cold comfort and I can offer no quick and easy remedies.

Please note that the C++0x improvements to the C++ features that most programmers see and directly use are unaffected. C++0x will still be a more expressive language than C++98, with support for concurrent programming, a better standard library, and many improvements that make it significantly easier to write good (i.e., efficient and maintainable) code. In particular, every example I have ever given of C++0x code (e.g., in "Evolving a language in and for the real world: C++ 1991-2006" at ACM HOPL-III, available at http://portal.acm.org/toc.cfm?id=1238844) that does not use the keywords "concept" or "requires" is unaffected. See also my C++0x FAQ (http://www.research.att.com/~bs/C++0xFAQ.html). Some people even rejoice that C++0x will now be a simpler language than they had expected.

"Concepts" were to have been the central new feature in C++0x for putting the use of templates on a better theoretical basis, for firming-up the specification of the standard library, and a central part of the drive to make generic programming more accessible for mainstream use. For now, people will have to use "concepts" without direct language support as a design technique. My best scenario for the future is that we get something better than the current "concept" design into C++ in about five years. Getting that will take some serious focused work by several people (but not "design by committee").

What Happened?

"Concepts," as developed over the last many years and accepted into the C++0x working paper in 2008, involved some technical compromises (which is natural and necessary). The experimental implementation was sufficient to test the "conceptualized" standard library, but was not production quality. The latter worried some people, but I personally considered it sufficient as a proof of concept.

My concern was with the design of "concepts" and in particular with the usability of "concepts" in the hands of "average programmers." That concern was shared by several members. The stated aim of "concepts" was to make generic programming more accessible to most programmers [BS&GDR2003], but that aim seemed to me to have been seriously compromised: Rather than making generic programming more accessible, "concepts" were becoming yet another tool in the hands of experts (only). Over the last half year or so, I had been examining C++0x from a user's point of view, and I worried that even use of libraries implemented using "concepts" would put new burdens on programmers. I felt that the design of "concepts" and its use in the standard library did not adequately reflect our experience with "concepts" over the last few years.

Then, a few months ago, Alisdair Meredith (an insightful committee member from the UK) and Howard Hinnant (the head of the standard library working group) asked some good questions relating to who should directly use which parts of the "concepts" facilities and how. That led to a discussion of usability involving many people with a variety of concerns and points of view; and I eventually — after much confused discussion — published my conclusions [BS2009].

To summarize and somewhat oversimplify, I stated that:

- "Concepts" as currently defined are too hard to use and will lead to disuse of "concepts," possibly disuse of templates, and possibly to lack of adoption of C++0x.
- A small set of simplifications [BS2009] can render "concepts" good-enough-to-ship on the current schedule for C++0x or with only a minor slip.

That's pretty strong stuff. Please remember that standards committee discussions are typically quite polite, and since we are aiming for consensus, we tend to avoid direct confrontation. Unfortunately, the resulting further (internal) discussion was massive (hundreds of more and less detailed messages) and confused. No agreement emerged on what problems (if any) needed to be addressed or how. This led me to order the alternatives for a presentation in Frankfurt:

- **Fix and ship**
  Remaining work: remove explicit "concepts," add explicit refinement, add "concept"/type matching, handle "concept" map scope problems
  Risks: no implementation, complexity of description
  Schedule: no change or one meeting
- **Yank and ship**
  Remaining work: yank (core and standard library)
  Risks: old template problems remain, disappointment in "progressive" community ("seven years of work down the drain")
  Schedule: five years to "concepts" (complete redesign needed) or never
- **Status quo**
  Remaining work: details
  Risks: unacceptable programming model, complexity of description (alternative view: none)
  Schedule: no change

I and others preferred the first alternative ("fix and ship") and considered it feasible. However, a large majority of the committee disagreed and chose the second alternative ("yank and ship," renaming it "decoupling"). In my opinion, both are better than the third alternative ("status quo"). My interpretation of that vote is that given the disagreement among proponents of "concepts," the whole idea seemed controversial to some, some were already worried about the ambitious schedule for C++0x (and, unfairly IMO, blamed "concepts"), and some were never enthusiastic about "concepts." Given that, "fixing concepts" ceased to be a realistic option. Essentially, all expressed support for "concepts," just "later" and "eventually." I warned that a long delay was inevitable if we removed "concepts" now because in the absence of schedule pressures, essentially all design decisions will be reevaluated.

Surprisingly (maybe), there were no technical presentations and discussions about "concepts" in Frankfurt. The discussion focused on timing and my impression is that the vote was decided primarily on timing concerns.

Please don't condemn the committee for being cautious. This was not a "Bjarne vs. the committee fight," but a discussion trying to balance a multitude of serious concerns. I and others are disappointed that

we didn't take the opportunity of "fix and ship," but C++ is not an experimental academic language. Unless members are convinced that the risks for doing harm to production code are very low, they must oppose. Collectively, the committee is responsible for billions of lines of code. For example, lack of adoption of C++0x or long-term continued use of unconstrained templates in the presence of "concepts" would lead to a split of the C++ community into separate subcommunities. Thus, a poor "concept" design could be worse than no "concepts." Given the choice between the two, I too voted for removal. I prefer a setback to a likely disaster.

Technical Issues

The unresolved issue about "concepts" focused on the distinction between explicit and implicit "concept" maps (see [BS2009]):

1. Should a type that meets the requirements of a "concept" automatically be accepted where the "concept" is required (e.g. should a type *X* that provides +, -, *, and / with suitable parameters automatically match a "concept" *C* that requires the usual arithmetic operations with suitable parameters) or should an additional explicit statement (a "concept" map from *X* to *C*) that a match is intentional be required? (My answer: Use automatic match in almost all cases).
2. Should there be a choice between automatic and explicit "concepts" and should a designer of a "concept" be able to force every user to follow his choice? (My answer: All "concepts" should be automatic.)
3. Should a type *X* that provides a member operation *X::begin()* be considered a match for a "concept" *C<T>* that requires a function *begin(T)* or should a user supply a "concept" map from *T* to *C*? An example is *std::vector* and *std::Range*. (My answer: It should match.)

The answers "status quo before Frankfurt" all differ from my suggestions. Obviously, I have had to simplify my explanation here and omit most details and most rationale.

I cannot reenact the whole technical discussion here, but this is my conclusion: In the "status quo" design, "concept" maps are used for two things:

- To map types to "concepts" by adding/mapping attributes and
- To assert that a type matches a "concept."

Somehow, the latter came to be seen an essential function by some people, rather than an unfortunate rare necessity. When two "concepts" differ semantically, what is needed is not an assertion that a type meets one and not the other "concept" (this is, at best, a workaround — an indirect and elaborate attack on the fundamental problem), but an assertion that a type has the semantics of the one and not the other "concepts" (fulfills the axiom(s) of the one and not the other "concept").

For example, the STL *input* iterator and *forward* iterator have a key semantic difference: You can traverse a sequence defined by *forward* iterators twice, but not a sequence defined by *input* iterators; e.g., applying a multi-pass algorithm on an input stream is not a good idea. The solution in "status quo" is to force every user to say what types match a *forward* iterator and what types match an *input* iterator. My suggested solution adds up to: If (and only if) you want to use semantics that are not common to two "concepts" and the compiler cannot deduce which "concept" is a better match for your type, you have to say which semantics your type supplies; e.g., "my type supports multi-pass semantics." One might say, "When all you have is a 'concept' map, everything looks like needing a type/'concept' assertion."

At the Frankfurt meeting, I summarized:

1. Why do we want "concepts"?
   To make requirement on types used as template arguments explicit
   - Precise documentation
   - Better error messages
   - Overloading

Different people have different views and priorities. However, at this high level, there can be confusion — but little or no controversy. Every half-way reasonable "concept" design offers that.

2. What concerns do people have?
   Programmability
   Complexity of formal specification
   Compile time
   Runtime

My personal concerns focus on "programmability" (ease of use, generality, teachability, scalability) and the complexity of the formal specification (40 pages of standards text) is secondary. Others worry about compile time and runtime. However, I think the experimental implementation (ConceptGCC [Gregor2006]) shows that runtime for constrained templates (using "concepts") can be made as good as or better than current unconstrained templates. ConceptGCC is indeed very slow, but I don't consider that fundamental. When it comes to validating an idea, we hit the traditional dilemma. With only minor oversimplification, the horns of the dilemma are:

- "Don't standardize without commercial implementation"
- "Major implementers do not implement without a standard"

Somehow, a detailed design and an experimental implementation have to become the basis for a compromise.

My principles for "concepts" are:

- **Duck typing**
  The key to the success of templates for GP (compared to OO with interfaces and more)
- **Substitutability**
  Never call a function with a stronger precondition than is "guaranteed"
- **"Accidental match" is a minor problem**
  Not in the top 100 problems

My "minimal fixes" to "concepts" as present in the pre-Frankfurt working paper were:

- **"Concepts" are implicit/auto**
  To make duck typing the rule
- **Explicit refinement**
  To handle substitutability problems
- **General scoping of "concept" maps**
  To minimize "implementation leakage"
- **Simple type/"concept" matching**
  To make vector a range without redundant "concept" map

See BS2009 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf).

No C++0x, Long Live C++1x

Even after cutting "concepts," the next C++ standard may be delayed. Sadly, there will be no C++0x (unless you count the minor corrections in C++03). We must wait for C++1x, and hope that 'x' will be a low digit. There is hope because C++1x is now feature complete (excepting the possibility of some national standards bodies effectively insisting on some feature present in the formal proposal for the standard). "All" that is left is the massive work of resolving outstanding technical issues and comments.

A list of features and some discussion can be found on my C++0x FAQ (http://www.research.att.com/~bs/C++0xFAQ.html). Here is a subset:

- atomic operations
- auto (type deduction from initializer)
- C99 features
- enum class (scoped and strongly typed enums)
- constant expressions (generalized and guaranteed; *constexpr*)
- defaulted and deleted functions (control of defaults)
- delegating constructors
- in-class member initializers
- inherited constructors
- initializer lists (uniform and general initialization)
- lambdas
- memory model
- move semantics; see *rvalue* references
- null pointer (*nullptr*)
- range for statement
- raw string literals
- template alias
- thread-local storage (*thread_local*)
- unicode characters
- uniform initialization syntax and semantics
- user-defined literals
- variadic templates

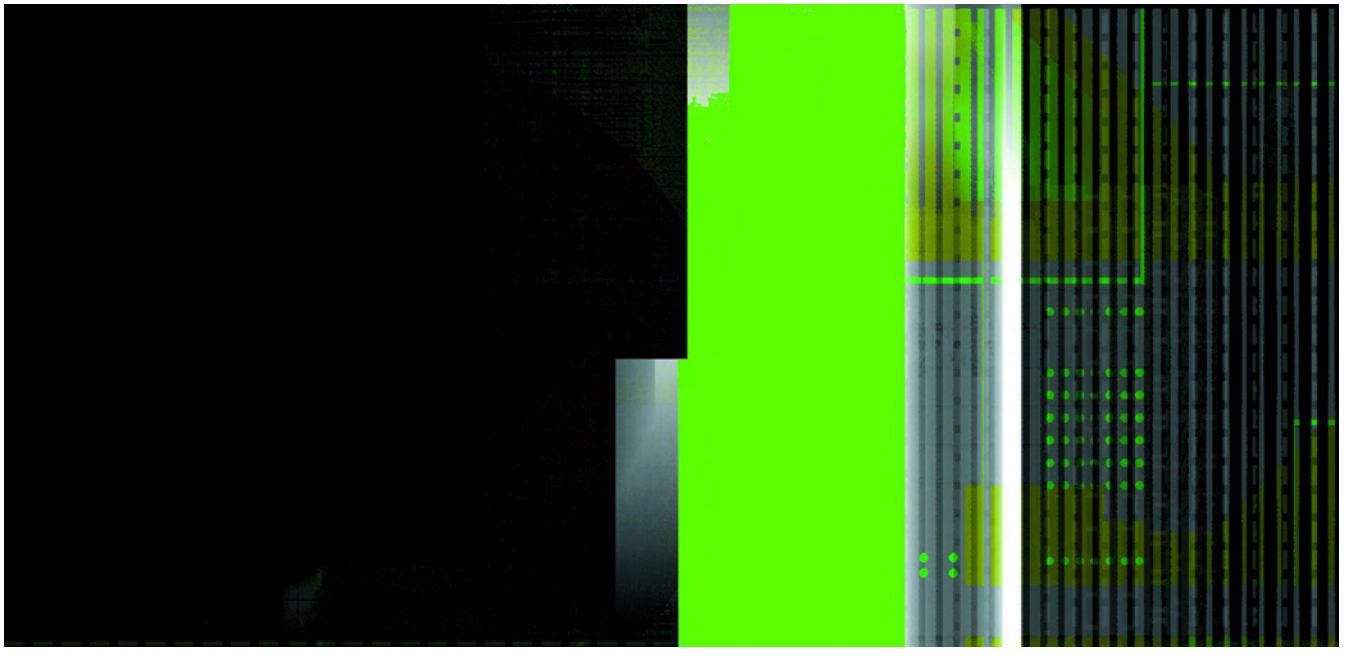
and libraries:

- improvements to algorithms
- containers
- duration and *time_point*
- function and bind
- *forward_list* a singly-liked list
- future and promise
- garbage collection ABI
- *hash_tables*; see *unordered_map*
- metaprogramming and type traits
- random number generators
- regex a regular expression library
- scoped allocators
- mart pointers; see *shared_ptr*, *weak_ptr*, and *unique_ptr*
- threads
- atomic operations
- tuple

Even without "concepts," C++1x will be a massive improvement on C++98, especially when you consider that these features (and more) are designed to interoperate for maximum expressiveness and flexibility. I hope we will see "concepts" in a revision of C++ in maybe five years. Maybe we could call that C++1y or even "C++y!".

—Bjarne Stroustrup designed and implemented the C++ programming language. He can be contacted at http://www.research.att.com/~bs/.

Return to Table of Contents



A Build System for Complex Projects: Part 1

A different approach to build systems

By Gigi Sayfan

Build systems are often a messy set of scripts and configuration files that let you build, test, package, deliver, and install your code. As a developer, you either love or loathe build systems. In this article series, I present a different approach to build systems, with the ultimate goal of completely hiding the build system from developers. But first, let me start with some personal history.

Early in my programming career I was a pure Windows developer (with the exception of my very first job, where I wrote Cobol programs for publishing Australia's Yellow Pages). While there was no build system to speak of, there was Visual Studio and Visual SourceSafe. I built Windows GUI clients, messed around with COM components, and picked up some nice C++ template tricks from ATL. And because automated unit testing wasn't very common back then, we created various test programs before passing code on to QA. This wasn't too painful since I worked for a small startup company and the projects weren't too big.

But I then moved to a company that developed software for chip fabrication equipment in the semi-conductor industry and BOOM! Life-critical and mission-critical real-time software running on six computers that controlled custom-built hardware in clean-room conditions. The software ran on several operating systems with about 50 developers contributing code. The development environment consisted of two machines running Linux and Windows/Cygwin. The deployment environment was Solaris and LynxOS RTOS. No more Visual Studio. After reading about 1000 pages of documentation in the first week and getting my .profile and .bashrc in order, I was assigned my first task — designing and implementing a build system to replace the existing one, which was a nasty combination of Makefiles and Perl scripts that actually worked but nobody was sure why (the original author had left the building). There were a few bugs (for example, the build system didn't always follow the proper dependency path) and a big requirements document. Clearly it would be impossible to evolve the current build system, so I had to create a new one from scratch. This was lucky because I had zero experience with Makefiles and Perl, coupled with the tolerance threshold of a Windows developer to gnarly stuff. I still have the same tolerance, but I now know something about Makefiles.

Some of the requirements were pretty unusual, like running a commercial code generator that produces code from UML diagrams on a Windows machine, then uses the artifacts to compile code on Linux, Solaris, and LynxOS. The bottom line is that I decided to take an unusual approach and wrote the entire system in Python. It was my first big Python project and I was really surprised at how well it went. I managed everything in Python. I directly invoked the compiler and linker on each platform, then the test programs, and finally a few other steps. For instance, I implemented friendly error messages that provided helpful suggestions for common errors (e.g., "FrobNex file not found. Did you remember to configure the FrobNex factory to save the file?").

While I was generally pleased with the system, it wasn't completely satisfactory. In lieu of Makefiles, I created build.xml files, a la Ant. That was a mistake. The XML files were verbose compared to Makefiles, big chunks were identical for many subprojects, and people had to learn the format (which was simple, but something new). I

wrote a script that migrated Makefiles to build.xml files, but it just increased code bloat. I created a custom build system without regard for the specific environment and its needs. I created a very generic system, with polymorphic tools that can do anything as long as you write the code for the tool and configure it properly. This was bad. Whenever someone says, "You just have to..." I know I'm in trouble. What I took away from this experience is that Python is a terrific language. It's really fun when you can actually debug the build system itself. Having full control over the build system is great, too.

Background: What Does a Build System Do?

The build system is the software development engine. Software development is a complex activity that involves tasks such as: source-code control, code generation, automated source code checks, documentation generation, compilation, linking, unit testing, integration testing, packaging, creating binary releases, source-code releases, deployment, and reports. That said, software development usually boils down to four main phases:

1. Developers write source code and content (graphics, templates, text, etc.)
2. The source artifacts are transformed to end products (binary executables, web sites, installers, generated documents)
3. The end products are tested
4. The end products are deployed or distributed

A good automated build system can take care of steps 2–4. The distribution/deployment phase is usually to a local repository or a staging area. You will probably need some amount of human testing before actually releasing the code to production. The build system can also help with that by notifying users about interesting events, such as successful and/or failed builds and providing debugging support.

But really, who cares about all this stuff? Actually everybody — developers, administrators, QA, managers, and even users. The developers interact most closely with the build system because every change a developer makes must trigger at least a partial build. When I say "developer" I don't necessarily mean a software engineer. I could be referring to a graphic artist, technical writer, or any other person that creates source content. When a build fails, it's most often because a developer changed something that broke the build. On rare occasions, it would be an administrator action (e.g., changing the URL of a staging server or shutting down some test server) or a hardware problem (e.g., source control server is down). A good build system saves time by automating tedious and error-prone activities.

Think about a developer manually building and unit testing a program. Without a build system, he has to very carefully build it properly, test it, and hand it over to QA. The QA person needs to run his own tests, then hand it to the administrator for deployment to a staging site, where more tests are run against the deployed system. If anything goes wrong in this process, someone must determine what happened. Automated build systems eliminate a whole class of errors. They never forget a step and they can pinpoint and resolve other errors by verifying that the source artifacts and intermediate artifacts are available and by scanning through log files and detecting failures.

Managers can also benefit from build systems. A passing build is the pulse of a project. If you have an automated build system with good test coverage (at the system level), managers can monitor project progress and be ready to release at each point. This in turn enables more agile development practices (if you are so inclined).

A build system can even help users in some cases. Think about systems that incorporate user-generated content and/or plug-ins. In most cases, you need to go over the content and ensure it doesn't break your system. A build system that automates some/all of these checks allows for shorter publish/release cycles for user-generated content.

Build System Problems

Okay, build systems are the greatest thing since Microsoft Bob (http://en.wikipedia.org/wiki/Microsoft_Bob). However, they still don't always live up to their potential:

- **They Don't Do Enough (Not Fully Automated).** This is one of the most common problems. A build system that is not fully automated can compile the software, create documentation, and package the final binary, but it requires a lot of user intervention to run various scripts, wait for previous stages to finish, check error reports, and so on.
- Requires a Lot of Discipline to Use Properly. Some build systems fail inexplicably if you don't follow a slew of obscure steps, like logging into the test server with a specific user, removing directory A, renaming directory B, making sure you perform step X only if the report generated by step Y says okay.
- **Requires Too Much Configuration.** Some build systems are very powerful and flexible, but are almost unusable due to excessive configuration. You have to define six different environment variables, modify three local config files, and pass eight different command-line options to the main build script. The end result is that 99% of the users use a single default configuration that probably doesn't fit their needs.
- **Caters Mainly To a Sole Stakeholder.** Another common problem is that a build system is often suitable for just one kind of stakeholder. For example, if the build system was developed mainly by the programmers who compile, link, and unit test all day, then the build system will have good support for these activities, but running integration tests or generating documentation may be poorly supported, if at all. On the other hand, if the build system was developed mainly by a release engineering team, then it will have good support for packaging final executables and will generate good reports about the percentage of passing test, but it may not be possible for developers to run just a single unit test and its dependencies, and they will either have to run the full-fledged build every time or hack the build system in a quick and dirty way (which might lead to errors).
- **Intractable Error Messages When Something Is Wrong.** Build systems perform many activities that involve external tools. The errors generated by these tools are often swallowed by the build system that much later generates

its own error message, which doesn't point to the root cause. This is a serious problem that hurts productivity and causes people to revert to manual but understandable build practices.

- **Inextensible and Undebuggable.** Franken-CodeBuild systems are often one of the earliest tools created at project initiation. The requirements of this early build system are usually minimal. As time goes by and the project grows, the demands from the build system grow too. Since the build system is an internal tool, less effort is dedicated to making it high quality code. More often than not, it is just a bunch of scripts slapped together and extended to support additional requirements by the tried and true practice of copy and paste. Such build systems quickly become a maintenance nightmare and can't be extended easily to accommodate new requirements.
- **Not Integrated With Developer's IDE.** Most build systems that don't come with an IDE built-in don't support IDEs. They are command-line based only and if a developer wants to work in an IDE, the IDE project files must be maintained and synchronized with the build system build files. For example, the build system may be Makefile-based, and a developer that uses Visual Studio has to maintain a .vcproj file for each project, and any additional files must be added to the Makefile as well.

The Perfect Build System

The build system I present in this series is open ended and can be used to automate any software process that is mainly file-based. However, the focus is on a cross-platform build system for large-scale C++ projects because these are often the most complicated to build. The perfect build system solves or minimizes the problems associated with existing build systems.

"Convention over configuration" is a principle that has successfully governed in domains like web frameworks, reducing the learning curve and increasing developer productivity. It demands that you organize your project in a consistent way (which is always good practice in any event):

The perfect build system solves or minimizes the problems associated with existing build systems

- **Regular directory structure.** This is the key principle on which the entire build system rests. Even in the most complicated systems, there is usually a relatively small high-level directory structure that contains a potentially huge number of similar directories. For example, a project may have a libs directory that contains all the C++ static libraries. The contents of the libs directory may grow and change, but it always contains a single type of entities.
- **Well-known locations.** The build system should be aware of the location and names of the top-level directories and "understand" what they mean. For example, it should know that the directories under libs generate static libraries that should later be linked into executables and dynamic libraries that depend on them.
- **Automatic discovery of files based on extension.** Each directory usually contains a small number of file types. Again, in the libs example, it should contain .h and .c/.cpp files and potentially a couple of other metadata files. The build system should know what files to expect and how to handle each file type. Once you have the regular directory structure in place, the build system "knows" a lot about your system and can do many tasks on your behalf automatically. In particular, it doesn't need in a build file in each directory that tells it what files are in it, how to build them, etc.
- **Capitalize on the small variety of subproject types.** In the C/C++ world, there are really only three types of subprojects: a static library, a dynamic library, and an executable. Static libraries (a compiled set of files bundled together) are the simplest. They are later linked into dynamic libraries and executables. Dynamic libraries and executables are similar from a build point of view. They both have source files and depend on precompiled static libraries to link against. It is important to build the dependent dynamic libraries and executables after building all the required static libraries. Many libraries (both static and dynamic) and executables use the same set of compiler and linker flags. Placing these groups under a parent directory informs the build system of these common flags and automatically builds all the subprojects.
- **Generate build files from templates for any IDE.** Different IDEs, as well as command-line based tools like Make, use different build files to represent the meta information needed to build the software. The build system I present here maintains the same information via its inherent knowledge combined with the regular directory structure and can generate build files for any other build system by populating the appropriate templates. This approach lets developers build the software via their favorite IDE (like Visual Studio) without the hassle involved in adding files, setting dependencies, and specifying compiler and linker flags.
- **Automatic dependency management based on *#include* analysis.** Managing dependencies can be simple or complicated depending on the project. In any case, missing a dependency leads to linking errors that are often hard to resolve. This build system analyzes the *#include* statements in the source files and recursively creates a complete dependencies tree. The dependencies tree determines what static libraries a dynamic library or executable needs to link against.
- **Automatic discovery of added/removed/renamed files and directories.** The regular directory structure, combined with knowledge of files types (e.g., .cpp or .h files), allows the build system to figure out what files it needs to take into account, so developers just need to make sure the right files are in the right directory.
- **Flexibility**

  Support static libraries, dynamic libraries, executables, and custom artifacts. All possible build artifacts are supported including custom ones like code generators, preprocessors, and documentation generators. The ability

to put similar files and subprojects under top-level directories in the regular directory structure is open to any subproject type.

Control the level of error messages. The build system is designed to support different users, such as QA, developers, and managers. Each type of user may be interested in different error messages.

Generate custom artifacts like language bindings. The build system is focused on building C/C++ code, but using the same practices and mechanisms it is possible to extend it to support additional artifacts, while maintaining all the existing benefits.

Allow overriding defaults. While the build system is intended to provide a hands-free experience, where all the necessary build information is derived automatically from the directory structure, it is possible to override it for special purposes, such as a single library that needs different flags.

- **Integrated Build System**

Build phases are executed from the same program. The build system is a cohesive program that operates on a set of templates and source files. This one-stop shop approach is very powerful for keeping the build process manageable.

Invoke external programs as a last resort. Ideally, the build system contains the entire logic of each build step. External programs are invoked only when the effort to implement the logic in the build system itself is deemed too costly. For example, the compiler and linker are invoked as external programs.

Full debugging of the build system. The fact that the build system is a single program allows users to debug the build process in real-time including setting breakpoints, viewing the current state, and finding live build system bugs. This is very different from standard declarative build files that usually only provide obscure error messages at a much later stage.

Hello, World (Platinum Enterprise Edition)

I hope you agree that this build system sounds awesome. But is it for real? To demonstrate and explore its capabilities, I will follow an imaginary software team that just started working on a new project.

The project is called "Hello, World!". The goal is to print it to the screen. To do this, over the course of this series, the team will create a complex project with multiple executables, static and dynamic libraries, and even Ruby bindings. The project will run on Windows, Linux, and Mac OS X. It will be built using a custom build system. To whet your appetite, here is a prototype in Python of the finished project:

```
print 'Hello, World!'
```

Project Kick-Off

Isaac, the sage development manager, assembled a team of brilliant software developers with umpteenth-years of experience in delivering high-performance enterprise applications. The kick-off meeting went well and the developers quickly reached a few decisions:

- The project will be developed mostly in C++,
- The system must be cross-platform and support Windows, Linux, and Mac OS X,
- The developers will be divided into four teams.

  Team H will develop a static library called libHello that returns "Hello".

  Team P will develop a dynamic library called libPunctuator that produces commas and exclamation points (and can be reused in future projects requiring punctuation).

  Team W will develop the complicated libWorld static library that must return the long and difficult word "World".

  Team U will develop an infrastructure project called libUtils that provides utility services to the other teams.
- The project will also deliver a Ruby language binding to make it more buzzword-compliant.
- The test strategy is to develop multiple test programs to test every library. Each team will be responsible for developing the test program for its library.
- The build system will be developed in Python by the renowned build expert Bob (aka "The Builder"). (No connection to Microsoft Bob, thank you.)

Bob carefully observed the source and required artifacts of the system and came up with the following directory structure. Each kind subproject is contained in a top-level directory under the source tree:

```
root
 |___ ibs
 |___ src
        |___apps
        |___bindings
        |___dlls
        |___hw (static libraries)
        |___test
```

- The *ibs* directory contains the files and templates of the build system. Note that it is completely separate from the source tree under *src*.
- The *src* directory contains all the source files of the system. Let's take a quick look at the top-level directories under *src*.

  apps. This directory contains all the (executable) applications generated by the system. Each application will have its own directory under *apps*.

  bindings. This directory will contain Ruby bindings at some point. At the moment it is empty.

  dlls. This directory contains the project's dynamic libraries.

  hw. This directory contains the project's static libraries. The reason it is called *hw* (as in "hello world") and not *libs* or a similar name is that it is very important to prevent name clashes with system or third-party static libraries. The automatic dependency discovery of the build system relies on analysis of *#include* statement. The unique hw part of the path of each static library allows unambiguous resolution of *#include* statements.

  test. This directory contains a subdirectory for each test program. Each test program is a standalone executable linked against the static libraries it is designed to test.

Next Time

In the next installment of this series, Bob and I delve into the innards of the build system and explain exactly how it works. Stay tuned.

— 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

Integrating jQuery Client-Side Data Templates With WCF

Using client-side templates and binding JSON data that's retrieved from a WCF service

By Dan Wahlin

In my article "Minimize Code by Using jQuery and Data Templates" (http://www.ddj.com/windows/217701311), I presented a JavaScript data binding template solution that I've been using to make it easy to bind JSON data to a client-side template without having to write a lot of JavaScript code. In this article, I demonstrate the fundamentals of using the client-side templates and binding JSON data that's retrieved from a WCF service. As a review (in case you didn't read the previous article), the template solution I've been using recently on a client project is based on some code written by John Resig (creator of jQuery), which is extremely compact. Here's a modified version of his original code that I wrapped with a jQuery extender:

```
$.fn.parseTemplate = function(data)
{
    var str = (this).html();
    var _tmplCache = {}
    var err = "";
    try
    {
        var func = _tmplCache[str];
        if (!func)
        {
            var strFunc =
            "var p=[],print=function(){p.push.apply(p,arguments);};" +
                        "with(obj){p.push('" +
            str.replace(/[\r\t\n]/g, " ")
               .replace(/'(?=[^#]*#&gt;)/g, "\t")
               .split("'").join("\\'")
               .split("\t").join("'")
               .replace(/&lt;#=(.+?)#&gt;/g, "',$1,'")
               .split("&lt;#").join("');")
               .split("#&gt;").join("p.push('")
               + "');}return p.join('');";

            //alert(strFunc);
            func = new Function("obj", strFunc);
            _tmplCache[str] = func;
        }
        return func(data);
    } catch (e) { err = e.message; }
    return "&lt; # ERROR: " + err.toString() + " # &gt;";
}
```

The *parseTemplate* method can be applied against a client-side template like the one below. Notice that the template is wrapped in a script block with the type set to *text/html* so that it isn't rendered by the browser. JSON properties are written out by using the <#= #> syntax and the template engine has full support for embedded JavaScript code.

```
<script id="MyTemplate" type="text/html">
    <table style="width:400px;">
        <thead>
            <tr>
                <th>First Name</th>
                <th>Last Name</th>
                <th>Address</th>
            </tr>
        </thead>
        <tbody>
        <#
            for(var i=0; i < d.length; i++)
            {
               var cust = d[i];
        #>
                <tr>
                   <td id="CustomerRow_<#= i.toString() #>">
                        <#= cust.FirstName #>
                   </td>
                   <td>
                        <#= cust.LastName #>
                   </td>
                   <td>
                      <#= cust.Address.Street #>
                      <br />
                      <#= cust.Address.City #>,
                         <#= cust.Address.State #>
                        <#= cust.Address.Zip #>
                   </td>
                </tr>
        <#
            }
        #>
        </tbody>
    </table>
    <br />
    <#= d.length #> records shown
</script>
```

This template outputs a simple table like Figure 1. Sure, I could have generated the table using DOM manipulation techniques, but being able to tweak a data template is much easier and productive in my opinion.

To use the template you'll need to have some JSON data available. Here's an example of creating JSON by hand and binding it to the template using the *parseTemplate* method shown earlier. The data returned from the template data binding operation is passed to the *html* method of the target *div*, which displays the data in the browser. Note: I'm defining the *d* property in the JSON object because WCF uses that name by default when it returns serialized JSON data.

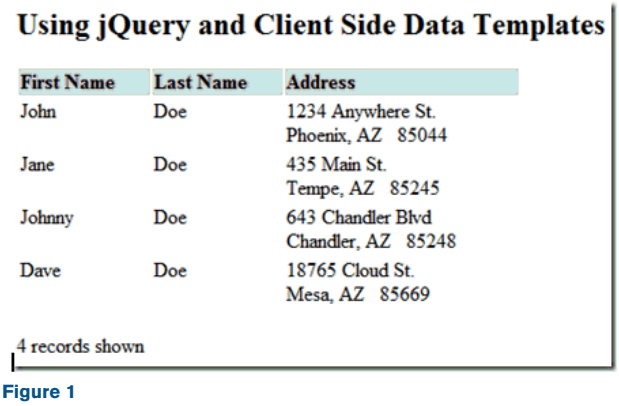


Figure 1

```
var json =
        {
            "d":
            [
               { "FirstName": "John", "LastName": "Doe",
                 "Address":
                  { "Street": "1234 Anywhere St.", "City": "Phoenix",
                    "State": "AZ", "Zip": 85044 }
               },
               { "FirstName": "Jane", "LastName": "Doe",
                  "Address":
                  { "Street": "435 Main St.", "City": "Tempe",
                    "State": "AZ", "Zip": 85245 }
               },
               { "FirstName": "Johnny", "LastName": "Doe",
                 "Address":
                  { "Street": "643 Chandler Blvd", "City": "Chandler",
                    "State": "AZ", "Zip": 85248 }
              },
               { "FirstName": "Dave", "LastName": "Doe",
                 "Address":
                  { "Street": "18765 Cloud St.", "City": "Mesa",
                    "State": "AZ", "Zip": 85669 }
               }
            ]
        };
var output = $('#MyTemplate').parseTemplate(json);
$('#MyTemplateOutput').html(output);
```

Of course, in the real-world you'll probably get the JSON data from some type of service (WCF, ASMX, REST, etc.). Here's a WCF service that returns a *List* of *Customer* objects and converts them to JSON. The service has the client script behavior enabled so that serialization from CLR objects to JSON objects occurs behind the scenes automatically.

```
[ServiceContract(Namespace = "http://www.thewahlingroup.com")]
[AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Allowed)]
public class CustomerService
{
    [OperationContract]
    public List<Customer> GetCustomers()
    {
        return new List<Customer>
        {
            new Customer {FirstName="John",LastName="Doe",
                          Address=
                           new Address{
                           Street="1234 Anywhere St.",
                           City="Phoenix",State="AZ", Zip=85044}},
            new Customer {FirstName="Jane",LastName="Doe",
                          Address=
                           new Address{
                           Street="435 Main St.",
                           City="Tempe",State="AZ", Zip=85245}},
            new Customer {FirstName="Johnny",LastName="Doe",
                          Address=
                          new Address{
                          Street="643 Chandler Blvd",
                          City="Chandler",State="AZ", Zip=85248}},
            new Customer {FirstName="Dave",LastName="Doe",
                          Address=
                          new Address{
                           Street="18765 Cloud St.",
                           City="Mesa",State="AZ", Zip=85669}}
        };
    }

}
```

jQuery's *ajax* method can then be used to call the WCF service and retrieve the data (jQuery provides other methods such as *getJSON* that could be used too if desired):

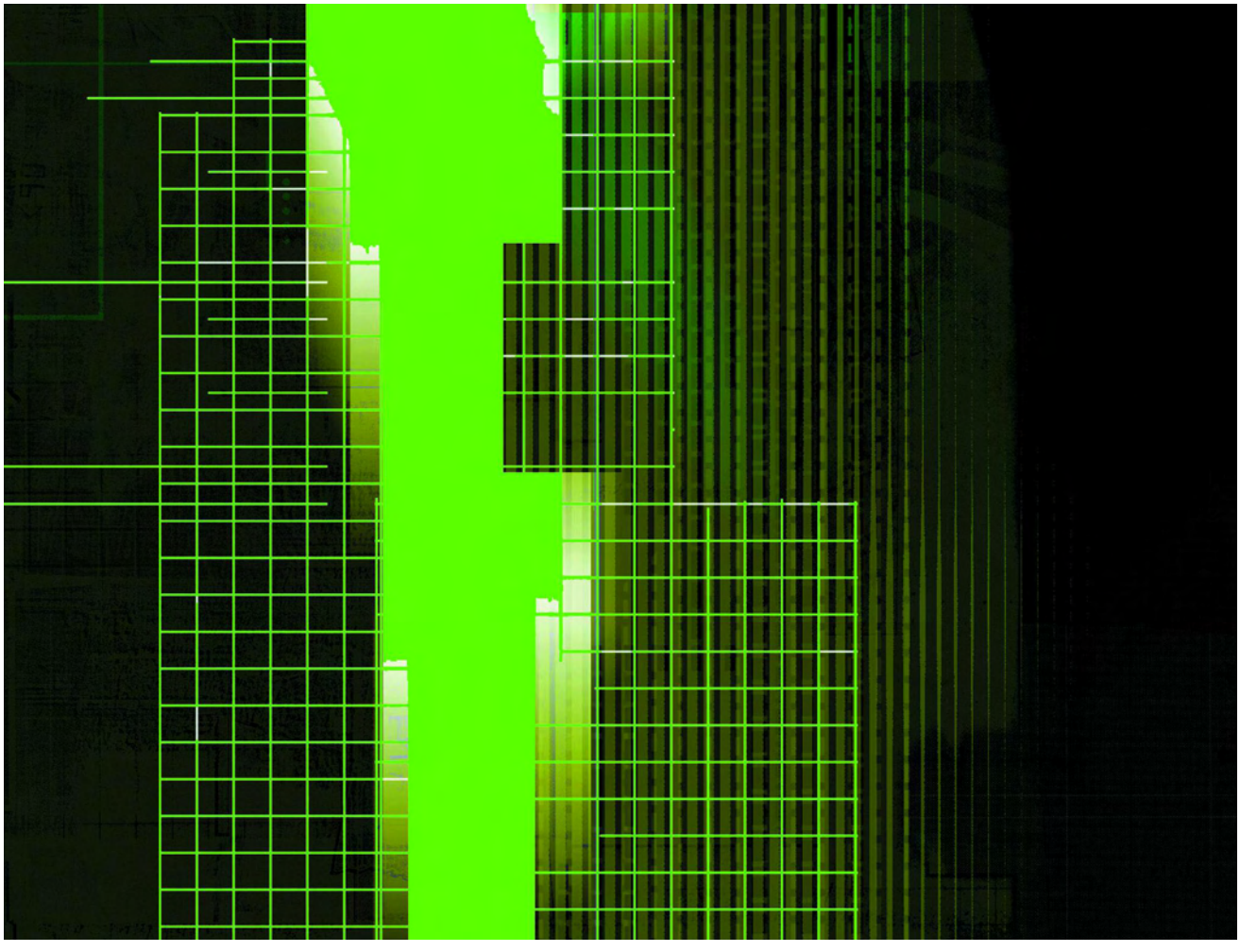
```
$.ajax(
{
     type: "POST",
     url: "CustomerService.svc/GetCustomers",
     dataType: "json",
     data: {},
     contentType: "application/json; charset=utf-8",
     success: function(json)
     {
          var output = $('#MyTemplate').parseTemplate(json);
          $('#MyTemplateOutput').html(output);

          //Add hover capabilities
          $('tbody > tr').bind('mouseenter mouseleave', function()
          {
               $(this).toggleClass('hover');
          });
     }
});
```

This code defines the type of operation, service URL to call, any data passed to the service, the content type, and a success callback. Once the service call returns, the JSON data is bound to the template shown earlier by locating the area where the template should be rendered to (*MyTemplateOutput* in this example) and then calling *parseTemplate*. Hover capabilities are also added using jQuery's *bind* method to highlight rows as the user moves the mouse in and out of them.

You can see that the amount of custom JavaScript that has to be written is kept to a minimum by combining jQuery with the client-side template, which ultimately leads to easier maintenance down the road. This is just one of several different client-side template solutions out there. ASP.NET 4.0 will also include a custom client-side template solution as well once released. You can download the sample code here (http://i.cmpnet.com/ddj/images/article/2009/code/jQueryDataTemplates.zip).

— Dan Wahlin (Microsoft Most Valuable Professional for ASP.NET and XML Web Services) is the founder of The Wahlin Group (www.TheWahlinGroup.com), which provides .NET, SharePoint, and Silverlight consulting and training services. Dan blogs at http://weblogs.asp.net/dwahlin.

Return to Table of Contents



Of Interest

JetBrains has released **Version 1.0 of MPS** (short for "Meta Programming System"), a language workbench and IDE for extending existing languages and creating custom Domain Specific Languages. By using MPS and DSLs created with its help, domain experts can solve domain-specific tasks easily, even if they're not familiar with programming. MPS is freely available, with a major part of its source code open and available under the Apache license, and can be downloaded bug-tracking system, code-named Charisma, is developed entirely with MPS. This issue tracker is a modern Web 2.0 application. To create it, a whole stack of web application languages was created: languages for HTML templates, controllers, database access, JavaScript, etc. MPS doesn't use any parsers. It works with the abstract syntax tree directly, so it doesn't require any parsing. Compiler construction knowledge might be useful, but you don't have to be an expert in this field in order to use MPS: it contains a predefined set of languages with which users can create their own languages. **http://www.jetbrains.com/mps/?mps1pr**

Intel has made available for free download **Prototype Edition 3.0 of the Intel C++ STM Compiler.** (STM is short for "Software Transactional Memory.") The Transactional Memory C++ language constructs that are included open the door for users to exercise the new language constructs for parallel programming, understand the transaction memory programming model, and provide feedback on the usefulness of these extensions with Intel C++ STM Compiler Prototype Edition. This posting includes the Intel C++ STM Compiler Prototype Edition 2.0 and runtime libraries for Intel transactional memory language construct extensions. **http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/**

TeamDev has released **Selenium Inspector,** an open-source library that runs on top of the Selenium, a tool designed to simplify automated testing of Web components, pages and applications — especially those written using JSF. The Selenium Inspector API lets you create testing solutions for variety of HTML rendering frameworks like JSF component libraries, Spring MVC, and Struts. Web developers can create object-oriented testing APIs for any Web UI library. The Java API for inspecting OpenFaces components is already included. Selenium Inspector provides an API similar to that of Selenium, but is simpler to use in many cases and provides a bit higher level of abstraction. It doesn't replace Selenium, but provides an additional API that you can use if you find it more appropriate for your actual needs. You can use both Selenium and Selenium Inspector APIs at the same time. **http://seleniuminspector.org/**

Three Key Challenges to Adding Parallelism to Your Applications

CLICK SCREEN TO LAUNCH VIDEO ABOUT 3 KEYS TO MULTICORE
For more videos on this topic, go to www.ddj.com//go-parallel/

Static analyzers try to find weaknesses in other programs that could be triggered accidentally or exploited by intruders. A report from the **National Institute of Standards and Technology (NIST)** entitled "**Static Analysis Tool Exposition (SATE)**, "edited by Vadim Okun, Romain Gaucher, and Paul Black, documents NIST's Static Analysis Tool Exposition —an exercise by NIST and static analyzer vendors to improve the performance of these tools. The static analyzers (and languages) in the study included Aspect Security ASC 2.0 (Java), Checkmarx CxSuite 2.4.3 (Java), Flawfinder 1.27 (C), Fortify SCA 5.0.0.0267 (C, Java), Grammatech CodeSonar 3.0p0 (C), HP DevInspect 5.0.5612.0 (Java), SofCheck Inspector for Java 2.1.2 (Java), University of Maryland FindBugs 1.3.1 (Java), and Veracode SecurityReview (C, Java). According to NIST's Vadim Okun, SATE was a long-overdue idea. "Most modern software is too lengthy and complex to analyze by hand," says Okun. "Additionally, programs that would have been considered secure ten years ago may now be vulnerable to hackers. We're trying to focus on identifying what in a program's code might be exploitable." While the SATE 2008 process was not designed to compare the performance of participating tools, it was successful in understanding some of their capabilities in a wide variety of weaknesses. SATE demonstrated that results from multiple tools can be combined into a single database from which further analysis is possible. While the backtrace explanations were useful, the study concluded that the evaluation might have been more efficient and less error-prone by closely integrating with the navigation and visualization capabilities of the tools. The SATE report is available at http://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf

Return to Table of Contents

Q&A: Open Database

What does the future hold for MySQL?

by Jonathan Erickson

Michael "Monty" Widenius was the creator of the MySQL database, and founder of Monty Program Ab. He recently spoke with Dr. Dobb's editor-in-chief Jonathan Erickson

Dr. Dobb's: What's the Open Database Alliance?
Widenius: The Open Database Alliance is a vendor neutral consortium of vendors and individuals commercially supporting or delivering services around MariaDB and MySQL. Open Database Alliance partners will support each other's open source initiatives, and resell each other's services. This makes it possible for customers to get all services they require around their database issues through any vendor in the Alliance.

Dr. Dobb's: What's MariaDB?
Widenius: It's a community developed branch of MySQL with bug fixes and new features developed by the MariaDB community, of which Monty Program Ab is an active member. We will keep MariaDB in sync with MySQL development to ensure all bug fixes and features in MySQL also exists in MariaDB. At this time MariaDB 5.1 should be notable faster, have more features and have fewer bugs than the corresponding MySQL 5.1 release.

Dr. Dobb's: Is SQL adequate for 21st century computing?
Widenius: Yes. SQL will be around for a long time because it's a very expressive language that is very easy to embed in web-based applications. As long as people are developing web pages with programming languages like PHP, Perl, Ruby, and Java, SQL will have it's place.

Dr. Dobb's: What will the biggest change in data storage in five years?
Widenius: SSD (solid-state drive) memory will force a paradigm shift in how data is stored and accessed and a lot of old proven database algorithms have to be changed because there is no seek time anymore.

Dr. Dobb's: What's the most exciting development in DBMS technology today?
Widenius: : On the software side, the usage of Memcached and Gearman to do inexpensive "cloud like" computing is of course interesting. We are also seeing dedicated inexpensive machines that provides Memcached interfaces which will notable speed up and simplify any setup that uses Memcached (which is a standard component for most busy web sites).

Dr. Dobb's: Will operating systems ultimately be successful in converting their filesystems into SQL-managed organizations of data?
Widenius: I think that is a stupid idea. Most data people store is not suitable really suitable for SQL. SQL will only notable slow things down when accessing things and will create a lot more fragmentation compared to modern file systems without providing anything really critical for the end user. Another problem is that SQL managed data is very bad of application that wants to have their own access to the part of the data (like another database server running on a SQL managed filesystem).

Return to Table of Contents

Using Google App Engine

Reviewed by Mike Riley

Using Google App Engine
by Charles Severance
O'Reilly Media
262 pages; $29.99

Even though Google App Engine has been available to developers for some time, deep technical books on applying this scalable cloud service have been sparse. Does O'Reilly's entry, under their Google Press imprint, fill this void? Read on to find out.

Like many web-connected developers, I have been aware of Google App Engine (GAE) since its invitation-only beta days, but never really took much interest in it. I am a big fan of the Python scripting language, but the fact that GAE uses Python as its preferred logic language somehow failed to grab me. One of the main reasons for this was at the same time as GAE's initial public beta, I was busy immersing myself in the Python-based Django framework and I wasn't about to confuse myself with an alternative approach to Python-centric web application development. Fortunately, GAE was constructed with enough flexibility to allow a framework like Django to live within its constructs, as detailed in this April 2008 article by Googler Damon Kohler (http://code.google.com/appengine/articles/django.html). Additionally, with the inclusion of Java support, GAE offers plenty of flexibility for the developer seeking a hosted cloud solution. Unfortunately, author Charles Severance failed to explore either of these important features in *Using Google App Engine*. The book is instead oriented toward first-time web programmers unfamiliar with even the most rudimentary aspects of web development. Nearly half the book is spent on the basics of HTML, CSS, HTTP and basic Python syntax. Considering the book's brevity and cost, this expenditure left few pages solely dedicated to the GAE.

Once the beginner tutorials of basic web page construction and delivery are out of the way, the second half of the book dives into a high-level overview of the GAE, its use of templates (based on Django's template system, no less), handling cookies and session management, using the proprietary GAE Datastore for structured data storage, creating a GAE account, uploading and testing a GAE-friendly application and understanding and fine-tuning the GAE memory cache parameters. Four appendixes, one each dedicated to the target development OS (Windows XP, Windows Vista, Mac OSX and Linux) literally repeat the same information with the name of the OS replaced and other minor differences. These were quite frankly a waste of paper; the author should have consolidated the OS variations into a simple grid or footnotes where appropriate. That would have left more space for explaining the inner workings, design optimizations and best practices for developing best of breed GAE designs.

Besides the minimal amount of unique material in the book, one of its biggest failings for me was presenting me with a convincing argument to use GAE in the first place. The advantages mentioned by the author read like a Google advertisement from a Google fanboy. The author failed to share any well-known websites that run on GAE, interview others who are as enamored with GAE as he is, provide a chapter or appendex on answering privacy and intellectual property right concerns, service level expectations, etc.. While the book did a fair job at elevating my interest in GAE, it wasn't enough for me to consider placing any of my web apps into Google's cloud.

Overall, O'Reilly (and Google for that matter) missed a golden opportunity with this book to deeply explore the technical facets of GAE. Instead, they spent paper and ink on rehashing basic concepts that have been much better served in other O'Reilly titles. While this might be a helpful book for someone who has no web development experience whatsoever, yet aspires toward understanding MVC patterns and a broader grasp of the complexity associated with modern day web applications, it's a major letdown for practiced developers seeking deeper understanding of GAE in the real world. Perhaps O'Reilly will revisit this technology again under a "Mastering Google App Engine" title.

Return to Table of Contents

Design for Manycore Systems

Why worry about "manycore" today?

By Herb Sutter

Dual- and quad-core computers are obviously here to stay for mainstream desktops and notebooks. But do we really need to think about "many-core" systems if we're building a typical mainstream application right now? I find that, to many developers, "many-core" systems still feel fairly remote, and not an immediate issue to think about as they're working on their current product.

This column is about why it's time right now for most of us to think about systems with lots of cores. In short: Software is the (only) gating factor; as that gate falls, hardware parallelism is coming more and sooner than many people yet believe.

Recap: What "Everybody Knows"

Figure 1 is the canonical "free lunch is over" slide showing major mainstream microprocessor trends over the past 40 years. These numbers come from Intel's product line, but every CPU vendor from servers (e.g., Sparc) to mobile devices (e.g., ARM) shows similar curves, just shifted slightly left or right. The key point is that Moore's Law is still generously delivering transistors at the rate of twice as many per inch or per dollar every couple of years. Of course, any exponential growth curve must end, and so eventually will Moore's Law, but it seems to have yet another decade or so of life left.

Mainstream microprocessor designers used to be able to use their growing transistor budgets to make single-threaded code faster by making the chips more complex, such as by adding out-of-order ("OoO") execution, pipelining, branch prediction, speculation, and other techniques. Unfortunately, those techniques have now been largely mined out. But CPU designers are still

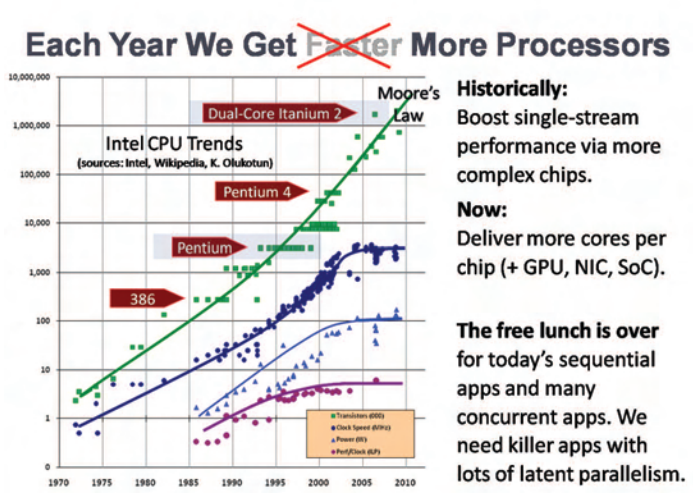


Figure 1: Canonical "free lunch is over" slide. Note Pentium vs. dual-core Itanium transistor counts.

reaping Moore's harvest of transistors by the boatload, at least for now. What to do with all those transistors? The main answer is to deliver more cores rather than more complex cores. Additionally, some of the extra transistor real estate can also be soaked up by bringing GPUs, networking, and/or other functionality on-chip as well, up to putting an entire "system on a chip" (aka "SoC") like the Sun UltraSPARC T2.

How Much, How Soon?

How quickly can we expect more parallelism in our chips? The naïve answer would be: Twice as many cores every couple of years, just continuing on with Moore's Law. That's the baseline projection approximated in Figure 2, assuming that some of the extra transistors aren't also used for other things.

However, the naïve answer misses several essential ingredients. To illustrate, notice one interesting fact hidden inside Figure 1. Consider the two highlighted chips and their respective transistor counts in million transistors (Mt):

- 4.5Mt: 1997 "Tillamook" Pentium P55C. This isn't the original Pentium, it's a later and pretty attractive little chip that has some nice MMX instructions for multimedia processing. Imagine running this 1997 part at today's clock speeds.
- 1,700Mt: 2006 "Montecito" Itanium 2. This chip handily jumped past the billion-transistor mark to deliver two Itanium cores on the same die. [1]

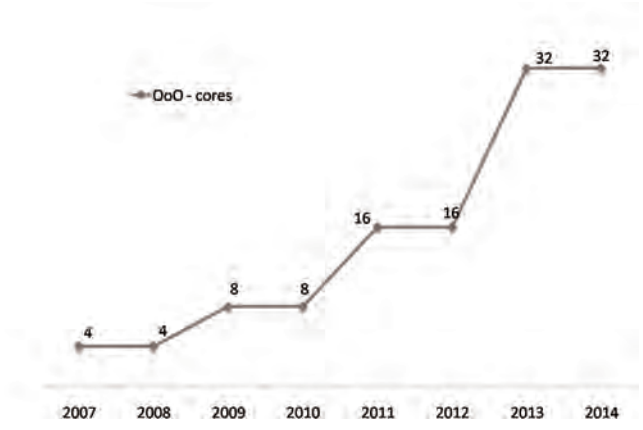


Figure 2: Simple extrapolation of "more of the same big cores" (not counting some transistors being used for other things like on-chip GPUs, or returning to smaller cores).

So what's the interesting fact? (Hint: 1,700 ÷ 4.5 = ???.)

In 2006, instead of shipping a dual-core Itanium part, with exactly the same transistor budget Intel could have shipped a chip that contained 100 decent Pentium-class cores with enough space left over for 16 MB of Level 3 cache. True, it's more than a matter of just etching the logic of 100 cores on one die; the chip would need other engineering work, such as in improving the memory interconnect to make the whole chip a suitably balanced part. But we can view those as being relatively 'just details' because they don't require engineering breakthroughs.

Repeat: Intel could have shipped a 100-core desktop chip with ample cache — in 2006.

So why didn't they? (Or AMD? Or Sun? Or anyone else in the mainstream market?) The short answer is the counter-question: Who would buy it? The world's popular mainstream client applications are largely single-threaded or nonscalably multi-threaded, which means that existing applications create a double disincentive:

- They couldn't take advantage the extra cores, because they don't contain enough inherent parallelism to scale well.
- They wouldn't run as fast on a smaller and simpler core, compared to a bigger core that contains extra complexity to run single-threaded code faster.

Astute readers might have noticed that when I said, "why didn't Intel or Sun," I left myself open to contradiction, because Sun (in particular) did do something like that already, and Intel is doing it now. Let's find out what, and why.

Hiding Latency: Complex Cores vs. Hardware Threads

One of the major reasons today's modern CPU cores are so big and complex, to make

Hardware threads are important, but only for simpler cores

Hardware threads have acquired a tarnished reputation. Historically, for example, Pentium hyperthreading has been a mixed blessing in practice; it made some applications run something like 20% faster by hiding some remaining memory latency not already covered in other ways, but made other applications actually run slower because of increased cache contention and other effects. (For one example, see [3].)

But that's only because hardware threads are for hiding latency, and so they're not nearly as useful on our familiar big, complex cores that already contain lots of other latency-hiding concurrency. If you've had mixed or negative results with hardware threads, you were probably just using them on complex chips where they don't matter as much.

Don't let that turn you off the idea of hardware threading. Although hardware threads are a mixed bag on complex cores where there isn't much remaining memory latency left to hide, they are absolutely essential on simpler cores that aren't hiding nearly enough memory latency in other ways, such as simpler in-order CPUs like Niagara and Larrabee. Modern GPUs take the extreme end of this design range, making each core very simple (typically not even a general-purpose core) and relying on lots of hardware threads to keep the core doing useful work even in the face of memory latency.

single-threaded applications run faster, is that the complexity is used to hide the latency of accessing glacially slow RAM — the "memory wall."

In general, how do you hide latency? Briefly, by adding concurrency: Pipelining, out-of-order execution, and most of the other tricks used inside complex CPUs inject various forms of concurrency within the chip itself, and that lets the CPU keep the pipeline to memory full and well-utilized and hide much of the latency of waiting for RAM. (That's a very brief summary. For more, see my machine architecture talk, available on Google video. [2])

So every chip needs to have a certain amount of concurrency available inside it to hide the memory wall. In 2006, the memory wall was higher than in 1997; so naturally, 2006 cores of any variety needed to contain more total concurrency than in 1997, in whatever form, just to avoid spending most of their time waiting for memory. If we just brought the 1997 core as-is into the 2006 world, running at 2006 clock speeds, we would find that it would spend most of its time doing something fairly unmotivating: just idling, waiting for memory.

But that doesn't mean a simpler 1997-style core can't make sense today. You just have to provide enough internal hardware concurrency to hide the memory wall. The squeezing-the-toothpaste-tube metaphor applies directly: When you squeeze to make one end smaller, some other part of the tube has to get bigger. If we take away some of a modern core's concurrency-providing complexity, such as removing out-of-order execution or some or all pipeline stages, we need to provide the missing concurrency in some other way.

But how? A popular answer is: Through hardware threads. (Don't stop reading if you've been burned by hardware threads in the past. See the sidebar "Hardware threads are important, but only for simpler cores.")

Toward Simpler, Threaded Cores

What are hardware threads all about? Here's the idea: Each core still has just one basic processing unit (arithmetic unit, floating point unit, etc.) but can keep multiple threads of execution "hot" and ready to switch to quickly as others stall waiting for memory. The switching cost is just a few cycles; it's nothing remotely similar to the cost of an operating system-level context switch. For example, a core with four hardware threads can run the first thread until it encounters a memory operation that forces it to wait, and then keep doing useful work by immediately switching to the second thread and executing that until it also has to wait, and then switching to the third until it also waits, and then the fourth until it also waits — and by then hopefully the first or second is ready to run again and the core can stay busy. For more details, see [4].

The next question is, How many hardware threads should there be per core? The answer is: As many as you need to hide the latency no longer hidden by other means. In practice, popular answers are four and eight hardware threads per core. For example, Sun's Niagara 1 and Niagara 2 processors are based on simpler cores, and provide four and eight hardware threads per core, respectively. The UltraSPARC T2 boasts 8 cores of 8 threads each, or 64 hardware threads, as well as other functions including networking and I/O that make it a "system on a chip." [5] Intel's new line of Larrabee chips is expected to range from 8 to 80 (eighty) x86-compatible cores, each with four or more hardware threads, for a total of 32 to 320 or more hardware threads per CPU chip. [6] [7]

Figure 3 shows a simplified view of possible CPU directions. The large cores are big, modern, complex cores with gobs of out-of-order execution, branch prediction, and so on.

The left side of Figure 3 shows one possible future: We could just use Moore's transistor generosity to ship more of the same — complex modern cores as we're used to in the mainstream today. Following that route gives us the projection we already saw in Figure 2.

But that's only one possible future, because there's more to the story. The right side of Figure 3 illustrates how chip vendors could swing the pendulum partway back and make moderately simpler chips, along the lines that Sun's Niagara and Intel's Larrabee processors are doing.

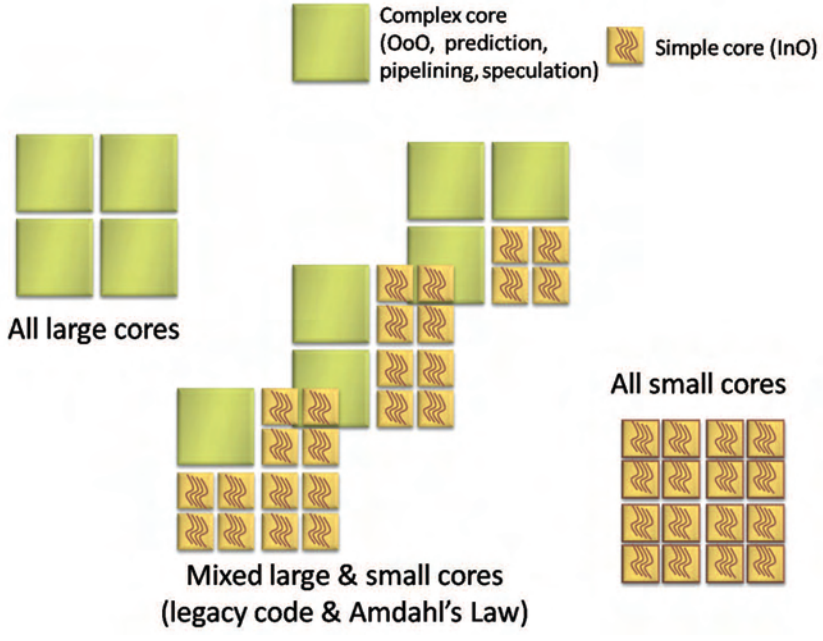


Figure 3: A few possible future directions.

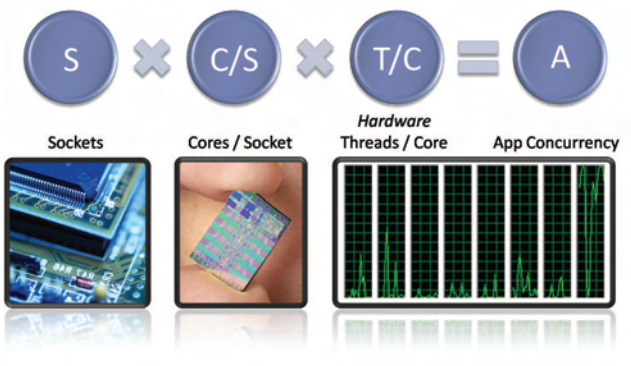


Figure 4: How much concurrency does your program need in order to exploit given hardware?

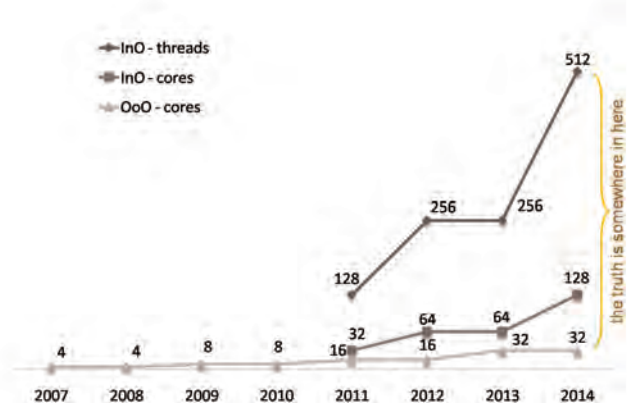


Figure 5: Extrapolation of "more of the same big cores" and "possible one-time switch to 4x smaller cores plus 4x threads per core" (not counting some transistors being used for other things like on-chip GPUs).

In this simple example for illustrative purposes only, the smaller cores are simpler cores that consume just one-quarter the number of transistors, so that four times as many can fit in the same area. However, they're simpler because they're missing some of the machinery used to hide memory latency; to make up the deficit, the small cores also have to provide four hardware threads per core. If CPU vendors were to switch to this model, for example, we would see a one-time jump of 16 times the hardware concurrency — four times the number of cores, and at the same time four times as many hardware threads per core — on top of the Moore's Law-based growth in Figure 2.

What makes smaller cores so appealing? In short, it turns out you can design a small-core device such that:

- 4x cores = 4x FP performance: Each small, simple core can perform just as many floating-point operations per second as a big, complex core. After all, we're not changing the core execution logic (ALU, FPU, etc.); we're only changing the supporting machinery around it that hides the memory latency, to replace OoO and predictors and pipelines with some hardware threading.
- Less total power: Each small, simple core occupies one-quarter of the transistors, but uses less than one-quarter the total power.

Who wouldn't want a CPU that has four times the total floating-point processing throughput and consumes less total power? If that's possible, why not just ship it tomorrow?

You might already have noticed the fly in the ointment. The key question is: Where does the CPU get the work to assign to those multiple hardware threads? The answer is, from the same place it gets the work for multiple cores: From you. Your application has to provide the software threads or other parallel work to run on those hardware threads. If it doesn't, then the core will be idle most of the time. So this plan only works if the software is scalably parallel.

Imagine for a moment that we live in a different world, one that contains several major scalably parallel "killer" applications — applications that a lot of mainstream consumers want to use and that run better on highly parallel hardware. If we have such scalable parallel software, then the right-hand side of Figure 3 is incredibly attractive and a boon for everyone, including for end users who get much more processing clout as well as a smaller electricity bill.

In the medium term, it's quite possible that the future will hold something in between, as shown in the middle of Figure 3: heterogeneous chips that contain both large and small cores. Even these will only be viable if there are scalable parallel applications, but they offer a nice migration path from today's applications. The larger cores can run today's applications at full speed, with ongoing incremental improvements to sequential performance, while the smaller cores can run tomorrow's applications with a reenabled "free lunch" of exponential improvements to CPU-bound performance (until the program becomes bound by some other factor, such as memory or network I/O). The larger cores can also be useful for faster execution of any unavoidably sequential parts of new parallel applications. [8]

How Much Scalability Does Your Application Need?

So how much parallel scalability should you aim to support in the application you're working on today, assuming that it's compute-bound already or you can add killer features that are compute-bound and also amenable to parallel execution? The answer is that you want to match your

application's scalability to the amount of hardware parallelism in the target hardware that will be available during your application's expected production or shelf lifetime. As shown in Figure 4, that equates to the number of hardware threads you expect to have on your end users' machines.

Let's say that YourCurrentApplication 1.0 will ship next year (mid-2010), and you expect that it'll be another 18 months until you ship the 2.0 release (early 2012) and probably another 18 months after that before most users will have upgraded (mid-2013). Then you'd be interested in judging what will be the likely mainstream hardware target up to mid-2013.

If we stick with "just more of the same" as in Figure 2's extrapolation, we'd expect aggressive early hardware adopters to be running 16-core machines (possibly double that if they're aggressive enough to run dual-CPU workstations with two sockets), and we'd likely expect most general mainstream users to have 4-, 8- or maybe a smattering of 16-core machines (accounting for the time for new chips to be adopted in the marketplace).

But what if the gating factor, parallel-ready software, goes away? Then CPU vendors would be free to take advantage of options like the one-time 16-fold hardware parallelism jump illustrated in Figure 3, and we get an envelope like that shown in Figure 5.

Now, what amount of parallelism should the application you're working on now have, if it ships next year and will be in the market for three years? And what does that answer imply for the scalability design and testing you need to be doing now, and the hardware you want to be using at least part of the time in your testing lab? (We can't buy a machine with 32-core mainstream chip yet, but we can simulate one pretty well by buying a machine with four eight-core chips, or eight quad-core chips… It's no coincidence that in recent articles I've often shown performance data on a 24-core machine, which happens to be a four-socket box with six cores per socket.)

Note that I'm not predicting that we'll see 256-way hardware parallelism on a typical new Dell desktop in 2012. We're close enough to 2011 and 2012 that if chip vendors aren't already planning such a jump to simpler, hardware-threaded cores, it's not going to happen. They typically need three years or so of lead time to see, or at least anticipate, the availability of parallel software that will use the chips, so that they can design and build and ship them in their normal development cycle.

I don't believe either the bottom line or the top line is the exact truth, but as long as sufficient parallel-capable software comes along, the truth will probably be somewhere in between, especially if we have processors that offer a mix of large- and small-core chips, or that use some chip real estate to bring GPUs or other devices on-die. That's more hardware parallelism, and sooner, than most mainstream developers I've encountered expect.

Interestingly, though, we already noted two current examples: Sun's Niagara, and Intel's Larrabee, already provide double-digit parallelism in mainstream hardware via smaller cores with four or eight hardware threads each. "Manycore" chips, or perhaps more correctly "manythread" chips, are just waiting to enter the mainstream. Intel could have built a nice 100-core part in 2006. The gating factor is the software that can exploit the hardware parallelism; that is, the gating factor is you and me.

Summary

The pendulum has swung toward complex cores nearly far as it's practical to go. There's a lot of performance and power incentive to ship simpler cores. But the gating factor is software that can use them effectively; specifically, the availability of scalable parallel mainstream killer applications.

The only thing I can foresee that could prevent the widespread adoption of many-core mainstream systems in the next decade would be a complete failure to find and build some key parallel killer apps, ones that large numbers of people want and that work better with lots of cores. Given our collective inventiveness, coupled with the parallel libraries and tooling now becoming available, I think such a complete failure is very unlikely.

As soon as mainstream parallel applications become available, we will see hardware parallelism both more and sooner than most people expect. Fasten your seat belts, and remember Figure 5.

References

[1] Montecito press release (Intel, July 2006) www.intel.com/pressroom/archive/releases/20060718comp.htm.

[2] H. Sutter. "Machine Architecture: Things Your Programming Language Never Told You" (Talk at NWCPP, September 2007). http://video.google.com/videoplay?docid=4714369049736584770

[3] "Improving Performance by Disabling Hyperthreading" (*Novell Cool Solutions* feature, October 2004). www.novell.com/coolsolutions/feature/637.html

[4] J. Stokes. "Introduction to Multithreading, Superthreading and Hyperthreading" (*Ars Technica*, October 2002). http://arstechnica.com/old/content/2002/10/hyperthreading.ars

[5] UltraSPARC T2 Processor (Sun). www.sun.com/processors/UltraSPARC-T2/datasheet.pdf

[6] L. Seiler et al. "Larrabee: A Many-Core x86 Architecture for Visual Computing" (*ACM Transactions on Graphics* (27,3), Proceedings of ACM SIGGRAPH 2008, August 2008). http://download.intel.com/technology/architecture-silicon/Siggraph_Larrabee_paper.pdf

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

[8] H. Sutter. "Break Amdahl's Law!" (*Dr. Dobb's Journal*, February 2008). www.ddj.com/cpp/205900309.

—Herb Sutter is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.

Return to Table of Contents