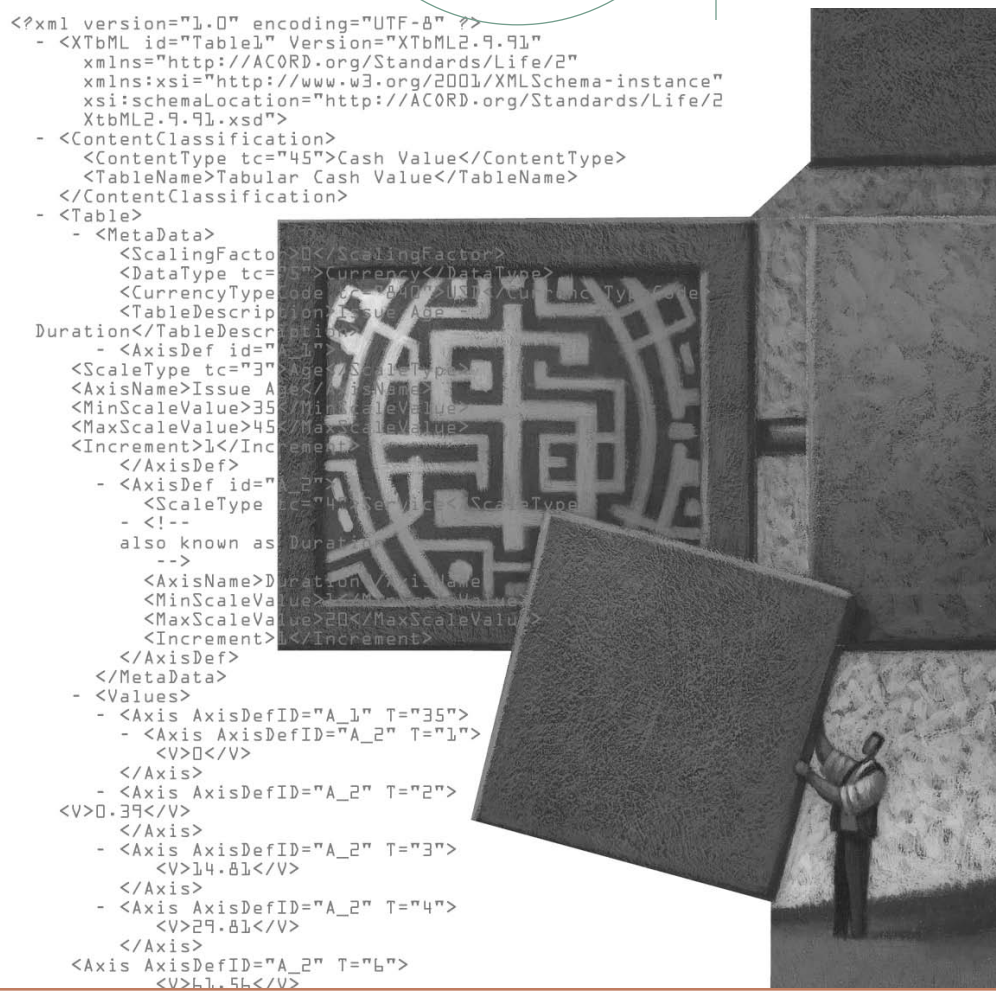
CompAct

TECHNOLOGY SECTION

"A KNOWLEDGE COMMUNITY FOR THE SOCIETY OF ACTUARIES"

Inside

Actuaries
The Best-Kept Secret in Business™



Computer Science Section Newsletter
Issue Number 18 • Volume 1
December 2005

Published quarterly by the Technology Section of the Society of Actuaries

475 N. Martingale Road, Suite 600
Schaumburg, IL 60173
phone: 847.706.3500
fax: 847.706.3599

World Wide Web: www.soa.org

Nariankadu D. Shyamalkumar
CompAct Editor
Assistant Professor
Statistics and Actuarial Science
241 Schaeffer Hall
The University of Iowa
Iowa City, IA 52242-1409
phone: 319.335.1980
fax: 319.335.3017
e-mail: *shyamal-kumar@uiowa.edu*

Articles Needed for the *CompAct Electronic Newsletter*

Your help and participation is needed and welcomed. All articles will include a byline to give you full credit for your effort. *CompAct* is pleased to publish articles in a second language if a translation is provided by the author. For those of you interested in working on *CompAct*, several associate editors are needed to handle various specialty areas such as meetings, seminars, symposia, continuing education meetings, new research and studies by SOA committees and so on. If you would like to submit an article or be an associate editor, please call Nariankadu Shyamalkumar, editor, at 319.335.1980.

CompAct is published as follows:

Publication Date	**Submission Deadline**
March 1	December 15

Preferred Format

In order to efficiently handle articles, please use the following format when submitting material:

Please e-mail your articles as attachments in either MS Word (.doc) or Simple Text (.txt) files. We are able to convert most PC-compatible software packages. Headlines are typed upper and lower case. Please use a 10-point Times New Roman font for the body text. Carriage returns are put in only at the end of paragraphs. The right-hand margin is not justified.

If you must submit articles in another manner, please call Joe Adduci, 847.706.3548 at the Society of Actuaries for assistance.

Please send electronic copies of the articles to:

N.D. Shyamalkumar
Technology Section Editor
e-mail: *shyamal-kumar@uiowa.edu*

Thank you for your help.

The Need for Speed

by Phil Gold

Today, actuarial modeling software requires blistering calculation speed to cope with the ever-increasing complexity of products and reporting bases, particularly deriving from risk management requirements and stochastic analysis.

This article will discuss how to keep up with the ever-increasing need for speed.

You can get speed in a stochastic actuarial model through three approaches: you can write fast code, you can use parallel processing techniques or you can introduce approximations. Approximations include simplified models, running fewer or representative scenarios, sampling, grouping and using less frequent time periods.

This article will focus on the first two of these approaches. If you use approximations, you will need to test your preferred approach against a complete calculation in order to calibrate your model, so you will still need an accurate baseline model on hand.

The further you can get using the first two techniques, the less use you will need to make of the various approximation methods. The objective is to make the software and hardware run fast enough that you don't have to make approximations.

Language

The first step is to choose an appropriate development language. Today's languages are mostly object oriented, and for very good reason. Object orientation allows you to build and maintain powerful complex systems, which would be almost impossible using traditional approaches.

C++, in the right hands, is capable of extremely high processing speeds, higher than other modern object-oriented languages. On the down side, C++ is extremely demanding of developers, and I would only recommend it for full-time professional programmers. Messing with pointers and multiple inheritance can be a double-edged sword—it can yield remarkable results, and it can get you into trouble very quickly. C++ is my personal recommendation for the professional developer. Not only does it offer the highest speed, but it also offers the greatest control over the environment.

Each language has its own strengths and weaknesses. Visual Basic allows you to get a lot of code written very quickly, but that code will not run at the same speed as C++. The good news is that some of the other modern languages such as Java, C# and VB.NET come with some pretty smart compilers which are reducing the performance gap between them and C++ and are well suited to smaller-scale development.

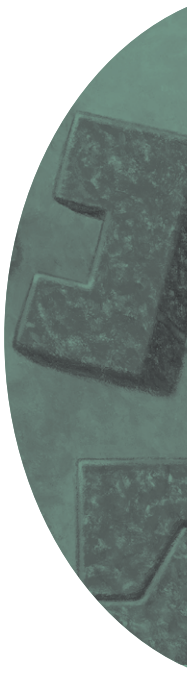
Database

The database engine will support both the input and the output of your model. There are many such engines on the market, and your first decision is whether to use an embedded engine, or to rely on an external relational database such as Oracle™, UDB™ or SQL Server™.

Extensive research reveals that there is simply no one database that does everything well. Each database engine is optimized for a different purpose. There are many benchmarks posted for the different engines on the market. You will find these on the sites of each major vendor.

(continued on page 4)

Phil Gold, ASA, FIA, MAAA, is a founding partner of GGY and chairperson of the Technology Section. He can be reached at pg@ggy.com.



Example

www.sleepycat.com/products/pdfs/wp_perf_0705c.pdf or

www.microsoft.com/presspass/press/2002/Dec02/12-18TPCBenchmarkPR.mspx.

These benchmarks may bear little relationship to the particular tasks for which you need the database engine. Those that look best on paper are often resource hogs and unreliable. Those that work best are often the ones with the longest history of development behind them, not necessarily the latest and greatest object databases. Preconceived ideas may be dangerous.

A word or two about XML: Although very useful for many different purposes, you will be best advised not to make any use of XML in time-critical processing because it is very wordy compared to regular Database RecordSets, and it takes longer to read and write than traditional methods.

Binary Storage

File storage is about the slowest thing you can do on a computer, so you should take steps to optimize it as much as possible. If you have to read or write data, it is much faster to do so in large blocks rather than field-by-field or record-by-record. Since you're extracting the data in blocks, you lose the ability to perform database operations within the database engine but you gain speed.

Data Compression

We normally think of data compression as trading speed for file size, but if you are careful, data compression can save you time as well as disk space. Consider a set of model projections; with maybe 100 lines being tracked for say 50 years monthly. If you can compress the data first, and then save it as a binary block, you will save a lot of hard disk writing time, often much more than the time it takes to do the compression.

Micro Optimization

If you are after the highest performance, you need to pay attention to the relative speed of those simple operations that may be called billions of times in your modeling run.

We all know that raising to the power is slower than multiplication. So your standard optimizations should always include replacing powers by multiplications, and multiplications by additions wherever possible.

There is a big payoff from fundamental optimizations to the root and power functions, which are used extensively in interest rate conversions. If the calculation you are trying to perform is the same as one you have recently performed, then by caching the previous inputs and outputs to the power function, you can save time. Your cache can be a simple one-element structure or a complex one with an extensive history. Alternatively you can use maps or lookup tables, or write your own optimized power and root algorithms.

Order of Calculation

Each way to arrange the loop, for records, scenarios and time periods has its advantages and disadvantages. Some are faster, some are more memory efficient and some work better with certain product features or matching strategies. You may end up supporting multiple methods.

Memory

Memory is faster than disk, so a good deal of optimization work is to replace disk read with memory reads. The danger in this approach is that your model quickly becomes a memory hog. A great deal of attention to detail is needed so that memory is allocated to the most significant items.

The most successful techniques for optimizing performance through the use of memory involve caching, where you devote a specific amount of memory for a given purpose. For example, you might dedicate 10 megabytes for holding recently used tables. Other techniques involve the use of global and static memory. If you use memory in this way, make sure the arrays are set up and refreshed at the right time.

Profiling

When you have your model programmed, you can see how fast it runs. A profiler, such as Metroverks Code Warrior™ or Compuware DevPartner Studio™ can calibrate your software to show how many times each procedure or even each line of code is called, and estimate how much time is spent executing each procedure or line of code. You can see where the bottlenecks are and where you should concentrate your efforts to improve the algorithms. Some profilers are much better than others, and none are perfect. Some do a better job of estimating where the time is spent, some have much better user interfaces than others, some require special builds of your software to perform their magic.

Just in Time Initializations

Originally in the C language, there was a requirement that all the variables used by a function were to be declared at the start of that function. C++ by design permitted variables to be declared on just about any line, even nested at any level of indentation.

You can exploit this new C++ rule. For example some code may only become active when certain controlling assumptions are met. Therefore code nested three levels deep can include its own set of working variables nested at the same level. If the code is executed, the corresponding variables are initialized just in time; otherwise the initialization for those variables is skipped.

Object Sizes and Lifetimes

You can break objects into two categories—those that need time-consuming dynamic memory allocations, "fat" and those that do not, "thin".

You can use thin objects liberally throughout your program. Declare and use them at any point in your code and allow them to be created and destroyed frequently.

The fat objects need to be carefully managed so that you keep them for much longer durations. You should avoid any pass-by-value uses of fat objects since these perform wasteful memory allocations.

Accuracy of Calculations

We quickly forget that our actuarial assumptions are pretty rough—can we really estimate a lapse rate to more than two significant figures? So we should think carefully before using the slower double-precision variables. You may need them sometimes, but probably not as much as you think.

The Hardware

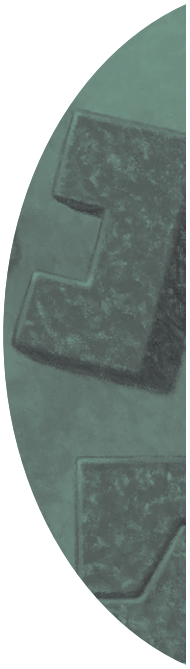
The next big thing, of course, is 64-bit architecture. Together with the appropriate software it offers improved access to large amounts of memory. We can expect some significant improvements from this source in the near future. The other development we are seeing on CPUs is dual core, and in the future, 4 or 8-core processors. Each core functions as an independent CPU.

Distributed Processing

Distributed processing can be used to accelerate a modeling system by running parallel calculations on multiple processors, in one box or over multiple boxes. Ideally if you have 20 processors, your program should run 20 times faster. The more research and development you do in this area, the closer you will come to this ideal. It takes a lot of work to provide a simple but fully scalable method for the users.

You need to think carefully about which parts of your software need the benefit of parallel processing. How are the various threads or processes going to communicate with each other? Remember that memory messaging is much faster than file messaging. You need to minimize network traffic and response times. Do you split the work by policy, by scenario or by some other characteristic? How do you minimize the overhang time when some processors are busy but others are finished? You also need to think about how to bring the results together from all the different processors.

(continued on page 6)



Just speeding up the calculations is only part of the job. You also have to make it reliable and easy to use. You need to develop fault-tolerance logic, because the user probably cannot afford the time to run the job again if a network connection goes down the first time, or if the cleaners turn off a machine they see left on after hours!

Dynamic load balancing is the process of making sure the work is being distributed optimally all the way through, even if conditions change during the run. This is complex to program, but will contribute significantly to scalability.

Grid Computing

As you add more and more machines to a distributed processing farm, the complexities of managing it become ever greater. The answer to this problem is grid computing, which automates the task of managing a farm so that many more processors can be used. Grid computing may handle many tasks, such as resource optimization, failure recovery, deployment and monitoring and can also support the sharing of a farm between multiple independent types of software.

As you scale up the number of processors, maintaining run-time scalability becomes tougher and tougher. Even more attention must be paid to error recovery and scheduling for multiple users. In this area it may pay to find partners with special expertise.

Admitting Mistakes

With the best will in the world, everyone makes mistakes. If it is a simple code error, you can fix it readily, but what if it is more basic than that? What if you have some of our fundamental architecture wrong? You can plow on regardless, you can go back and change the architecture or you can offer two different ways to run the software—the old way and the new way. The really big deal here is admitting the mistake—the bigger the mistake, the harder it may be to admit it. Admitting mistakes can be very expensive, but if you don't do it, your ultimate progress may be severely limited.

You won't necessarily find your mistakes unless you specifically look for them. This should be a continuous process. You need to set up a structure for peer review of the architecture and the code, regular regression testing and strictly enforced programming standards.

Pulling It All Together

Speed is not something you can eke out of a system after you have developed all the functionality. Rather you need to consider your speed target up front, since it can affect the programming language you use and the equipment you target.

Then you need to have speed in mind as a prime requirement all along the way, and you must be willing to go back and fix past mistakes, since no one has perfect foresight. It takes more than good decision-making and efficient code.

As your application develops, you need to use profiling tools to discover the roadblocks preventing your software from running at high speed, and you have to be willing to invest a lot of time and effort into fine details.

You need to research ways to introduce parallel processing into your code and take advantage of distributed processing and grid computing to scale up your application to meet the ever-increasing demands.