RECORD, Volume 31, No. 1*

New Orleans Life Spring Meeting
May 22–24, 2005

Session 53 OF
The Need for Speed: Achieving Maximum Run Time Performance

Tracks: Risk Management, Technology

Moderator: Francis P. Sabatini

Panelists: Philip Gold
Alex Korogodsky†

Summary: Risk management, valuation and product development activities have required increasingly complex financial models. These models could require seriatim processing, nested stochastic processes and thousands of scenarios. For many of these activities, run times can be measured in days and even months if the right technology is not applied. This session has practitioners discussing the ways they have addressed run time issues by leveraging technology, using thoughtful approaches to coding and model construction and other techniques. The presenters share their approaches to gaining speed. These include software and coding optimization, hardware optimization, distributed processing and parallel and grid computing techniques. The attendee receives an understanding of the need for carefully planned approaches to modeling and the creation of effective modeling environments.

MR. FRANCIS P. SABATINI: I've been doing risk management for a long time, and in the past several years I've learned to appreciate the need for speed. This will be an interesting topic and a unique combination of the Technology Section and the Risk Management Section, although most of the material is going to focus on the need for speed.

---

†Mr. Alex Korogodsky, not a member of the sponsoring organizations, is senior manager at Ernst & Young in New York, NY.

Note: The charts referred to in the text can be downloaded at: http://handouts.soa.org/conted/cearchive/NewOrleans-May05/053_bk.pdf.

Alex Korogodsky is a senior manager with Ernst & Young (E&Y) in our New York office. He is not an actuary. He has spent most of his career on the technology side, working at Prudential and then with E&Y, where he plays a very active role in defining technology solutions for E&Y in terms of our modeling environment as well as for many of our clients.

Phil Gold is vice president for R&D with GGY. He is a fellow of the Institute of Actuaries, has a master of arts degree, and he holds degrees in economics and operational research. He has worked in product development at Abbey Life in England, followed by a four-year stint at Manual Life in the group annuity and reinsurance areas. He became a vice president at the National Reinsurance Company of Canada. He is the incoming chair of the SOA Technology Section.

MR. ALEX KOROGODSKY: Performance optimization is dear to my heart because it's probably one of the most important things that we do right now for the firm, as well as for our clients. You know that the industry is intensively competitive. The products are more complex than ever. Volatility of the market reflects the pressure to grow GAAP earnings. All these challenges, in turn, are transmitted to the actuarial domain, and we're asked to do more work in less time with fewer people.

As an example of a typical computational task, I'd like to consider, say, a liability Greeks calculation via the overnight seriatim process. Basically the nature of this calculation is that you take 500,000 policies over a 20-year projection, you have 1,000 base scenarios, and then for every Greek you would have two additional scenarios. Shock up and shock down, and if, for instance, you're doing a delta, rho and gamma hedging, you wind up with 7,000 scenarios right there. When you try to estimate how long it would take to run through this stuff, you wind up with anywhere from 200 to 400 days of single-CPU run time.

The bottom line is that you can't get to solving this problem by an incremental advance in performance. You need a radical improvement, and that's what we want to talk to you about today. One of the traditional approaches to solving for this that we hear from some people is, "Let's not even bother. Forget about it. We'll manage without it." Another traditional approach is to go to your senior management and set the expectations by saying, "Well, I need a couple of weeks to run that model, and God forbid that I have an input error or any other issue with the model that I need to run. " As long as the management is willing to live with that type of approach, you might be fine.

Sacrificing on the model complexity—simplifying the models by reconfiguring and restructuring—is yet another approach to take. Some people throw additional computers into the run, manually divide up the work between those computers, manually push the "Run" button on each computer and then assemble the results at the end. That sort of works, as long as you have maybe seven to 10 machines. But I can hardly imagine how we can do it on hundreds of CPUs.

Apparently these approaches have a big advantage: there is no need for any sort of revolution here. It's no major change, right? That's an incremental change. But as such, it's still labor intensive. It's just an incremental improvement over what you have to date. Ultimately, it becomes a logistical nightmare.

Personally, my number one challenge to all of these approaches is, how do I explain to my family that I have to go to work on Sunday or Saturday and run the models? My wife doesn't want to live with that. She said, "You've got to do something."

Also, how do you manually start the work on hundreds of computers, and how do you manage that run? How do you manage the fact that you have a number of modelers competing for the same computing resources? They all want to use that CPU power that you have. How do you handle the failure? How uncommon is it for us to start a model on Friday night before leaving the office, hoping that by Monday it will be done? We come back on Monday and find out that it crashed only a half-hour after we left the office. That has happened a few times. How do we manage that? How do we utilize the computing power that our companies already have in place? There are hundreds and thousands of CPUs literally sleeping overnight, doing nothing or close to nothing. Is there a way to leverage that, especially during the peak times during the month end, quarter end, and year end?

Most important, as we start tapping into the advanced computational resources in our companies, how do we make IT folks comfortable? You all know that they can promote your initiative or they can kill it in about five minutes by coming up with 15 arguments why not to do things. All of these are challenges to solving the problem.

What we suggest is the framework that I want to set up today, which is to use a holistic approach. It's not a one-dimensional problem that we're trying to solve here. What we need is a robust computational infrastructure, a modeling infrastructure that can allow you to deliver order of magnitude improvements.

The four dimensions of this framework are optimized processes, optimized hardware, optimized model architecture and optimized model code. I'm going to take you through each one of these four dimensions in the next 10 or 15 minutes and talk a little bit about each and see what's involved. We do see in practice that this approach works and delivers order of magnitude improvement. However, we need to recognize that it's a transformational thing; it's not an incremental change. You need to rethink the way that you do modeling today in order to reap those benefits.

First is the optimized process dimension. This is everything that happens before and right after the calculation engine phase. All these things need to be optimized and processes made repeatable, auditable and, to the extent possible, automated. This is integrating everything that you serve to the model as an input: the data, in-force/new business, the assumptions, the product specifications. The focus is data

quality, because we see that if you spend time out front cleaning up, then you have to deal with fewer engine failures.

I had a conversation with one of my clients a couple of weeks ago who said, "Well, it's a great approach, and we like it a lot. Don't worry about the data quality—we've got it." I've been in the insurance industry too long to know that our data are not perfect. If you don't believe me, the session right across from us is talking for an hour and a half just about data quality. It's very important to isolate these problems early in the game, so that they don't lead you to catastrophic results on the engine.

Decision support capability is everything that happens to the model outputs after the engine's work. This mainly speaks to your ability to report, analyze and explain the results. Again, the idea with process optimization is that you remove any sort of process bottlenecks that stop you from being effective.

One other thing that is critical here and that we see as being more and more important is the model development life cycle. What this refers to is the second generation modeling platform, such as ALFA, MoSes and Prophet. All come with the ability for you to program in languages like C or C++. This is a serious programming environment, and software engineering learned a long time ago that you need to have productivity effects to manage this cycle. How do you develop the code? How do you manage the code? How do you deploy the code? How do you manage your test environment versus "product environment"? There are ways to do that. There is no need to reinvent the wheel. If we learn from the software engineering industry, we could adopt a few approaches that work very effectively in that. We don't need to be cutting and pasting the code, etc.

The second dimension is hardware optimization. The rationale there actually is that optimized hardware placed very well with the fourth dimension, which is the optimized model code. The notion here is that the more hardware optimization you do, the less you need to optimize the code and the less need you have for sitting there and figuring out what else you need to do to get to your desired run times?

When we talk about optimized hardware, there are two fundamentally interesting concepts that we need to consider. One is that you need to have a capability to analyze and test implications of different performance optimization techniques. It's almost like a test bed or a sandbox environment where you can experiment, which in all theory and practice should be independent from where you run your production models. So that you can afford to tie in those computing resources at the same time, you do need that capability of researching and diagnosing where the bottlenecks are.

In this production type of environment where you run your models, a great idea would be to run them remotely so that you connect to that environment, start the model, disconnect, do your other things, then connect later, get the results and

move on. You don't want to tie in, like we used to tie in, our consultants' laptops or employees' laptops or desktops with the model run. There is production infrastructure that can handle that.

One of the factors to consider in that regard would be, first of all, the resource utilization on your computer. Again, you need to understand how your model is using those resources, how it's using the CPU cycles and how it's using the memory. What's happening with the input/output operations? How often does it read and write to and from the disk? What are the network utilization patterns? Also, consider advances in microchip technology. Intel's Pentium 4, for instance, comes with a hyperthreading option, and you could squeeze some additional CPU cycles based on that technology. Then, of course, there's a big conversation about distributive environments. Distributive environments work great for a small number of computers, anywhere from 20 to maybe 40 to 50. When you start to reach 100 or so machines, the traditional distributive environment fails, and then you need to start thinking about utilizing grid-enabled computing, which helps with this multiple-user, multiple-resource job management and resource utilization.

The third dimension of the framework is the optimized model architecture. This is, for instance, the cell structure. Some models that I find in forecasting don't have to have the degree of detail that, for instance, the risk-related or capital-related modeling efforts have. Of course, all these second-generation modeling platforms come with a model building facility where you can do that right in the model. The question is, do you need to? Do you have to? Is it the most optimal way? We've seen time after time that people are starting to move this functionality out of the modeling engine into the tools that manage the data, the design to manage the data, and letting the engines do what they're best at, which is run the calculations. You need to think about whether this is the case that you'd like to try and decide.

Eliminating unnecessary output is another idea that could be used here. I am talking about a production model run where you don't need to write to the wrong log. This is number one. Most engines today will create the wrong log, and you don't need to have it done for production purposes. As you're developing the model, it's extremely important as you're trying to validate the model. But beyond that, for production runs it's an extreme.

In terms of estimating what columns you need, should you increase from the model, do you need all of them or maybe a subset? Do you need to use columns, or could you use fixed arrays? Things like that are important. If you have a top model with submodel structures, is it important for you to output everything from the submodels, or can you just limit the model output to the top model only? All these considerations play into, as I was saying before, what happens to my model output later. All this decision-support capability to analyze and explain the results, depending on what requirements you have for that, you would need to scale down on your output as part of the model architecture.

Then there are some other techniques, for instance, pregenerating static cash flows for products that don't have a dependency on economic scenarios. For example, do we need to calculate the reserves that require reprojections, or could you settle for something smaller than that or easier than that, such as cash surrender value for certain types of models? There are different tricks of the trade here. Some of the examples that I'm giving are just examples, and there's much more to it, of course.

The fourth dimension is code optimization. Again, this is the capability that becomes available with the second-generation tools, as they have what they call "open code." It's not the open sort of engine code; it's the open application code where you can create your models in C or C++ language. Naturally, the minute you start using your programming language like C or C++, you open a big opportunity door to code optimization, and we have decades of experience in optimizing the C code.

Let's talk about some of the most important things for us to worry about. First, do we access our multidimensional arrays in a row order? The reason for that concern is that they're stored in memory in row order, and if you were copying the columns, then you're creating a lot of inefficiencies for the CPU and memory allocation. There's a big topic that talks about loop optimization such as loop unrolling and removing invariant data so that you could do the calculations outside of the loop instead of sticking them in there.

There's also the matter of using the math functions. Phil is going to give you more details on how to deal with that, but as an example, obviously a power function is very CPU-expensive and, to the extent possible, if you can replace it with a multiplication function, that would be a way to go. The other rationale with math functions, for instance, is that you could look into replacing your division with multiplication, or you could be working with numbers and then divide them by 100 instead of using floating-point calculations. There are tricks like, for instance, using single quotes around your alphanumeric data instead of double quotes. All that optimizes the code, and, again, to the extent possible, even the 10, 15 or 20 percent performance that you can squeeze at this level will help.

The role of compilers is also important because in all this you used to be boxed in with a choice of one compiler. Our research shows that while some compilers are great in doing the model development and help you to quickly compile the models, they're providing the opposite effect during the run time. There are compilers where maybe it takes them longer to compile, but the run time is much faster. We were able to prove this point during some of our benchmark tests. We found out that one compiler, for instance, works from a run time perspective better than others. Intel works better than Microsoft. As a related topic, you could use special math libraries that are designed to be more efficient than native C++ calculations. Again, it's all connected to the notion of compilers, because typically these math libraries are provided to you as source code.

There are a couple of considerations that you need to keep in mind. First, performance depends on the type of the model and application. We're often asked, "How much faster can my MoSes model be or my ALFA model be?" That's an unfair question. I always tell my graduate students the perfect answer: It depends. Everything is so dependent on a particular application or on a particular model. But typically, you see the run time spiraling here. The question becomes, what was the coefficient of linearity? How linear is linear? Another consideration that has great importance for distributive computing is to recognize the fact that models respond differently to advancing the number of model cells versus advancing the numbers of scenarios.

I'll make a controversial statement here, which I know some of you might challenge. I believe that if you spend enough time on model optimization, you can get comparable run times no matter what tool you use. This especially refers to second-generation tools, where there are great capabilities in terms of hardware and language optimization. Probably the systems that are closed code and vendor-made might be optimized a little further than that. But if we're comparing apples to apples and just talk about the second-generation modeling platforms, I'm sure we can get to very comparable run times. The most important consideration is that you have to invest in studying the behavior of your model to diagnose where the bottom maxes out. What really happens with your model so that you can then address this for the dimensions in a most optimal way?

I'd like to say a word on grid computing now. In the last couple of SOA events, there have been a number of presentations on grid computing. Vendors tell you how great this approach is, and it is indeed great. You get an instant 10 or 20 percent performance gratification by deploying grid technology simply because you know the modeling tool vendors built this distributive computing functionality with their packages, but that's not what they're good at. Their niche is the actuarial calculation engine. The grid vendors, on the other hand, are very experienced in distributive computing. Needless to say, they are capable of creating much leaner distribution methodology.

The second consideration with grids is that if you try to benchmark whether or not to use the grid, comparing your distributive processing on five CPUs, five computers or 10 computers, and then running those five on five machines connected with the grid, you're not going to see a lot of performance improvement. The performance advantage you will notice is as you step into hundreds of CPUs, which is something that simple distributive processing just can't handle. This goes back to one of the challenges that I said up front in this presentation: How do you leverage the use of hundreds and thousands of sleeping CPUs in your company?

Scalability is another thing, which is how you merge this peak time need for computational resources. There's fault tolerance, which is how you protect from those issues related to a model crash. If one of the computational notes dies in the middle of the run, what does it mean for the whole run? Does it need to be

restarted? Hopefully not. Service levels allow you to schedule resources and runs much more flexibly. Most important, since you can't really deploy grids in your own data center within the actuarial department, IT people will challenge you. For what else can we leverage that technology for? The notion here is that you can leverage it to pretty much anything and everything, including, by the way, your Excel models if you still have those. You can grid-enable the Excel models. Security is a consideration.

Most of the real benefits are public; they've been advertised by the companies that have made strides with high-performance computing (HPC). I've decided not to give you the company names for obvious reasons, but if you're interested, I can refer you. Some of the things that people have reported is that you can truly go from a 10- to 100-CPU environment capability. Real-life clients report performance improvements such as going from 2.5 hours to 10 minutes, or from 18 hours to 32 minutes. This is drastic. This is the order of magnitude improvement that we seek.

Most important, people say that they can reverse this long-standing paradigm of actuaries messing around with data and pushing the buttons 80 percent of the time and interpreting the results 20 percent of the time. Now you can reverse that and spend most of your time analyzing the results, which is what you have been trained to do. People talk about delivering higher-quality services to clients faster than ever before. I'm sure your chief actuary would be happy to hear that. Ultimately, that leads to a stable environment. So the benefits are there. They've been recorded.

The last thing I want to talk about is resolving the considerations. I had an e-mail in my mailbox three months ago. I printed it out, and it's now hanging on top of my desk. The e-mail said, "Alex, we're ready to move on the high-performance computing environment. Where can we buy one?" Then it asked me how much it cost. Unfortunately, there is no cookie cutter here. You can't buy it; you need to build it. You need to invest time in this. You need to invest time in models, in restructuring and in rethinking the model architecture, because some models that we've seen are distributed out of the box, and some are not. That goes back to what I was saying about code optimization. If you can use vector arrays, they're very much distributed. If not, then they're not. You need to have the capability to diagnose your bottlenecks and how you optimize for speed. That capability has to be there. You need to solve the problem. Are you building your own mini-actuarial data center? Are you going to leverage the computing resources available to you in a company? You do need to have the adequate service level for CPU utilization, because if you are tapping into computers for people who don't have anything for modeling responsibilities and they want to run something on their machines and their CPU is 100 percent utilized, they're not going to be happy campers. You do need to somehow manage that.

The bottom line is that the market is heating up for this. There is a need. Frank and others talked about this yesterday during their sessions when they talked about hedging, C-3 Phase 2 and other computationally intensive modeling topics. They

said one thing at the end of the presentation: How do we deal with the run times? It's crazy what's going on. So there is a need there. Many actuarial software vendors are developing capability for that. In terms of your competition, a lot of our clients are already working on that or thinking about it, or they are already finished working on that. It is, I think, very critical for you to think of a strategy there.

MR. PHILIP GOLD: I'd like to begin with a few words of background. Like most of you, I imagine, I became an actuary because I wanted to work with computers. That was the promise I was given when I was hired at Abbey Life in England, and they were true to their word. In those days, I wrote pricing programs and rate manuals in FORTRAN. In 1980, I moved to Manual Life in Toronto and developed group pricing programs in APL before switching to financial reinsurance. In 1984, I became vice president of underwriting at National Reinsurance and somehow found time to develop a pricing system for reinsurance in Basic. Today, my company, GGY Inc., is in the business of writing actuarial pricing valuation of modeling software for the insurance industry and has been doing so for over 16 years. We employ around 30 developers, the majority of which are actuaries. I'm in charge of the R&D area. Our developers spend about a third of their time developing new functionality, a third of their time on checking and testing and a third of their time on optimizing the code for maximum performance and reliability. That allocation is rather unusual in the industry and reflects a long-term perspective.

Speed has always been vitally important to us, and we've developed quite a reputation in this area. We need the speed so that companies can use our software for seriatim or life-by-life valuation or seriatim-based modeling. Now that stochastic processing has become more and more important to our users, we've been placing an even greater priority on finding new ways to speed up our system. We have devoted a lot of time and effort to the task. The aim is to create a system capable of running stochastic seriatim modeling and valuation for large blocks of business in a 12-hour time period. If you can't run something overnight, then it's difficult to build a daily work schedule around it.

I'm going to take you through a number of concepts today, some of which are quite straightforward, some more esoteric, and some only for the seasoned C++ programmer, for which I apologize in advance. I'm a very practical fellow, and I'm not going to introduce any ideas that I haven't put into practice myself. This list is extensive, but each item I cover has already proved its worth in creating software to meet that ambitious target.

You can get speed through a number of approaches. You can write fast code, you can bring in parallel processing or you can make approximations. For those approximations, you can use simplified models, you can use fewer scenarios, you can use representative scenarios, you can do sampling of records or grouping of records and you can switch from accurate calculations to, say, quarterly or annual calculations instead. I'm not going to talk about approximations today. I'm going to focus my attention on the first two of these options. But if you're interested in

pursuing the approximations, you will need to do a lot of testing of your preferred approach against the full calculation in order to calibrate your model initially and then again from time to time. So you will still need a fast, 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 object in this game is to make the software and the hardware run fast enough so that you don't have to make many or any approximations. Today I'll do my best to cover the various techniques my company has researched and implemented to get these very long run times down to manageable proportions. If you're writing your own system, you can use some or all of these techniques. If you're buying software off the peg or there's already a software system in place, these ideas may provide a useful checklist.

First, you need to choose an appropriate development language. If you're trying to build a fast system, you'd better choose a fast language and compiler. In the past, the fastest language was Fortran. This may still be the fastest language for pure calculations, but it's pretty much a dead language these days. At one time, APL was a popular language for actuaries, but it's fallen out of favor today. Today's languages are mostly object-oriented, for very good reasons.

Object orientation allows you to build and maintain powerful, complex systems, which would be almost impossible using traditional approaches. When we first started, we were writing in Professional Basic. But we moved our calculation engine to C++ many years ago because our experiments with various languages showed us that C++ was extremely fast and well suited to our application. C++ in the right hands is capable of extremely high processing speeds, higher than any other modern object-oriented language. The emphasis is on the phrase "in the right hands," since C++ is extremely demanding of developers, and I would not recommend it for anyone other than full-time professional programmers. Messing with pointers and multiple inheritance can be a two-edged sword. It can yield remarkable results, and it can get you into trouble very quickly.

While C++ is appropriate for our shop, it may not be the right tool for your development. Each language has its own strengths and weaknesses. Visual Basic (VB), for example, allows you to get a lot of code written very quickly. But that code will not run at the same speed, nor will you have the same flexibility over the environment as the guys who program in C++. The good news is that some of the other modern languages, such as Java and 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 scales of development. Some people have had great success using Excel or other Microsoft office components as a base for their software.

When you choose your architecture, there are many choices to be made, depending on your objectives. Are you programming for yourself, or are there multiple end users? Is this a general-purpose system or a single-purpose tool? Does it have to

work just with your current products, or should it be capable of modeling any product you can think of? How should the end user gain access to leading edge product features? What's the life cycle of this product? Do you intend to maintain this system for the next 20 years or just the next six months? If you're in it for the long term, it makes sense to build a hierarchy of efficient classes and algorithms, which can take years to develop, and to come back to these classes from time to time and optimize them further with the aid of the latest techniques and hindsight. All of these considerations can affect the run speed of the software.

Consider the open code question. You may not have considered that open code can impose a large penalty on speed, but we found that closed or vendor-maintained code gives us the opportunity to fully optimize the code, both globally and through time, which would not be possible if our clients were also making changes or additions to the code. We chose the vendor-maintained approach to get the maximum possible speed. Of all the factors I'll talk about today, this is probably the single most significant contributor to the speed of our software.

Let's talk about databases. You probably need a database engine of some kind to support your calculations. You'll probably use it for both the input and the output of your model. There are many such engines on the market. Your first decision is whether to use an embedded engine or to rely on an external relational database, such as Oracle or SQL Server. In our case, we used a database engine of our own design for many years and then switched to the Jet database engine with our move to Windows. Jet has its limitations, but it is much faster than the SQL Server or Oracle for some operations, and you can address it directly with SQL from within C++.

The raw speed of a database engine is only one of the considerations. Connectivity is also important and must be matched to the needs of the intended users and the platforms on which they will run. Scalability to large file sizes, wide record sets and multiple users and processors are all-important qualities, while the single most important concern is robustness. How reliable is this engine under load? We've conducted a lot of research into database engines, and I can tell you with full confidence that there's simply no one database that does everything well. Each database engine is optimized for a different purpose. There are many data benchmarks posted for the different engines on the market. We found that these benchmarks bore little relationship to the particular tasks for which we would be needing the databases. We simply had to write our own test suites for the things that we felt were the most important.

We were quite stunned by the results of our experiments. The ones that looked the best on paper were often resource-hungry and unreliable. Those that worked best were often those with the longest history of development behind them, not necessarily the latest-and-greatest object databases. Throw your preconceptions out the window. In the end, we decided that our system could be considered as a number of separate modules and that our best strategy was to match each part of

the system to the database engine best suited to its requirements.

Now I'd like to say a word or two about Extensible Markup Language (XML). Although it's very useful for many different purposes, you'd be best advised not to make use of XML in time-critical processing because it is very wordy compared to regular database record sets, and it takes longer to read and write than traditional methods. Housing and conversions of data types can take a significant amount of time if you're working with millions of records, and it's best to remove these from the critical path.

We found that one way to speed up the calculations is to have all the data relationships in compiled source code. But that's an awkward and unreliable way to program a system for the long term. The alternative is to arrange these relationships in a database to be queried at run time. This database approach is fine for readability and long-term development, but at run time you have to query that database, and this slows down the system. To get the best of both worlds, we maintain all the relationships in databases, for which we have developed some very fancy visual editing tools, but we also write code generators that build code directly from those definitions in the database. The output from these code generations gets compiled with the other source code. It takes longer to build the system, but it runs much faster. Another advantage of this approach is that we can insert lots of validity tests into the code generator to make sure that the database relationships are consistent. If these tests fail, the system build fails too, and the inappropriate relationships can never reach production.

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's always much faster to do so in large blocks rather than field by field or record by record. You lose the ability to perform certain database operations, but you gain speed. Our developers chose carefully what will be a field in a RecordSet, and what will be stored as part of a large binary blob attached to that RecordSet for speed. If you do use binary storage techniques, then you'll need to offer utilities to give the users access to the original database fields on demand through some kind of import/export facility. Most developers do not bother with this dual approach since it complicates the programming and can lead to a duplication of data, but we found it to be of enormous benefit in speeding up stochastic processing and well worth the effort.

We normally think of data compression as trading speed for file size. If you're careful, data compression can save you time as well as disk space. Consider a large set of model projections, with maybe 100 lines being tracked for 50 years monthly. If you can compress the data first, perhaps by using some kind of PKZIP algorithm, and then save it as a binary block, you'll save a lot of hard disk writing time, often much more than the time it takes to do the compression. You lose the ability to retrieve individual values from the file because you can no longer locate to a specific field, so there are disadvantages, but for some kinds of information this is

not important. In our software, we use six different proprietary lossless compression algorithms just for storing projections, and we do save both a lot of time and a lot of disk space.

I'm going to talk about micro-optimization. If you're 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 a power is slower than multiplication, so your standard optimization should always include replacing powers by multiplications and replacing multiplications by additions wherever possible. We found an enormous payoff by fundamental optimizations to the root and power functions, which are used extensively in interest rate conversions. The best way is to rewrite the code so you don't have to call a power function, but if you can't eliminate it altogether, you may be able to add a run time test that only calls the power function if certain conditions are met. Then you can start developing caches.

A cache is a memory system that can store one or more previous results of a calculation or read request. When you call a cached function, it looks to see whether it already has the answer in memory before doing the calculation or read. If the calculations you're trying to perform are the same as ones you recently performed, by caching the previous inputs and outputs to the power function, you can save time. You cache can be a simple one-element structure or a complex one with an extensive history. Alternatively, you can use maps or lookup tables. In our code, we found that in some cases up to 40 percent of run time was spent inside rate conversion routines. By optimizing the code using advanced cascading caches, we brought the time down to 10 percent. We then reduced that to 3 percent through a proprietary technique we call "intelligent caching." We then reduced down to the 1-percent level through the use of some advance mathematics to replace the C++ *pow* function with a series of faster functions that work within specific ranges and through the use of a controller to arbitrate between them.

Let's talk about the order of calculations. Our first approach to stochastic processing was to put a big loop around our model for the number of scenarios to process. With minimal programming we could get the right answers. But that loop meant every policy had to be read in once per scenario. If you're running 1,000 scenarios, you're reading the assumptions 1,000 times. By offering an alternative order of processing so that the scenario loop comes within the policy loop, we were able to speed up the system dramatically. This was a lot more work for the developers, but it paid off big time. Each way to arrange the loops has its advantages and disadvantages, so you may end up supporting multiple methods.

Memory is faster than disk, so a good deal of optimization work is to replace disk reads with memory reads. The danger in this approach is that your model quickly becomes a memory hog, and this restricts the kind of hardware on which you can run it. You need a very big machine. A great deal of attention to detail is needed so that memory is allocated to the most significant items. We try to avoid any

structure that makes the amount of memory required depend on the number of records to process or the number of scenarios, since this limits the size of your model. For some orders of calculation we can achieve this, while for others you may need more memory if you increase the number of scenarios. In such cases, you should give tools to the user to manage the memory. Show the user how many scenarios he or she can run or how many time periods he or she can include on a given set of hardware.

The most successful techniques for optimizing performance through the use of memory involve caching. We devote a specific amount of memory to a given purpose. For instance, you might dedicate 10 megabytes for holding recently used tables. You need to do a lot of experimentation to optimize cache design and sizes. Other techniques involve the use of global and static memory, and, if you use memory in this way, you have to be very careful to make sure that the arrays are set up and refreshed at the right time.

Now we get to one of the most esoteric items. In a complex system, you may be reading the product features many times even for a single policy. For example, the premiums for valuation purposes may not be the same as the actual product premiums. If you're running four reserves and a projection basis, that can mean reading the premiums five times. Even if the commission scale doesn't change, you may need to recalculate the commissions because they depend on the premiums that may have changed. So we developed a set of dependency switches and an in-memory message board to keep track of all these things on a dynamic basis. When you calculate the tax reserves, for example, the system will only prepare the premium vector if it has changed since the last basis. It will only calculate commissions if the premium scale has just changed and so on. The ultimate aim of this message board system is that no assumption needs to be looked up unless it has changed, whether it's between reserve bases, between scenarios or between model points. This is pretty advanced optimization. It took a lot of care to set up initially, but now it works well for us with very little maintenance.

For those of you familiar with Developer Studio, this technology may be similar to the incremental compile feature, which means that in a big project, only some of the source files have changed since your last compile. Just those files will be compiled this time around. That, of course, makes a very big difference to the speed of computation.

Let's talk about some dynamic optimization methods. Ideally, the code you run through on any run should be just those lines that are necessary to produce the output required. For example, if you're trying to calculate a return on investment that doesn't depend on required surplus, then skip the required surplus calculation. One solution is to have different compiled code for each possible requirement and to run the code that best fits the requirement of this run. If you allow a lot of choice of outputs, then this may lead to a large number of executables, which may be too much to manage. Another way is to have a system of flags that direct traffic

through the code, avoiding the unnecessary routines. This is an area in which you can invest a lot of time to get a rich payoff. It may also require you to reorder your code. A third, more advanced, way is to make use of C++ virtual functions. In this case, at run time you would create an instance of an object from a short list of different types, according to the purpose of this particular run. Then you process it through one or more virtual functions. Each type of object may implement the same virtual function in a different way, some faster than others. By this technique, you've transformed a static code convenience, the virtual function, into a dynamic speed optimization.

When you have your model programmed, you can see how fast it runs. But with the help of a profiler, you can see where the bottlenecks are, where the most time is spent and how often each routine is called. Unfortunately, profilers affect the run speed of the program they're measuring. Rather like trying to view an electron, you bump into Heisenberg's uncertainty principle, so the timing results are often distorted and misleading, although the number of calls reported should be accurate. Some profilers allow you to set the unit of time measurement, and we found that this is a useful tool. Without naming names, some profilers are much better than others, and none are perfect. We supplement our profiler of choice by the addition of some home-built tools that can be applied to particular sections of code to give much more accurate diagnostics.

I'm going to get into some specific C++ optimizations. Native C++ arrays are fast, but they're inflexible and bug-prone. They do not indicate out-of-bounds conditions. They can't be dynamically resized, and they are always zero-based. Not everything is zero-based out there in real life. Bounds checking is vital, at least in debug builds, in getting robust code. So we looked for a set of safe, flexible vector classes that we could use instead. We were not happy with anything commercially available, so we built our own, not just one set, but several, each customized to a particular purpose. One set of classes provides fixed-size arrays. These provide identical performance to the built-in C arrays, but provide such object features as subscript checking in debug builds and the ability to retrieve both data and dimensions. These sets of vectors avoid any memory allocations. Another set of classes provides resizable arrays. Indexing speed is not quite as fast as the fixed size arrays, but it's very close. These classes provide intelligent memory allocations that recycle dead blocks from previous allocations.

Another aspect is decomposition, which Alex referred to before but not by that name. Since two-dimensional indexing of arrays is slower than vector indexing, all of the vector classes support the feature of decomposition. A two-dimensional array can, without any memory overhead, decompose into individual one-dimensional vectors. It's very handy to grab a one-dimensional slice of a two-dimensional array just prior to a loop and index on that within the loop. It can also speed up the code calculation. You can write code in C++ that's compiler switch dependent. So for debug builds, we've built classes with full bound checking and diagnostics. For release builds, we drop this excessive baggage.

The Need for Speed: Achieving Maximum Run Time Performance 16

The goal of C++ was to provide a great many high-level abstractions. The central idea was to create a nice programming environment for the programmer. The C language, its precursor, took the opposite approach. It provided features only if they could be implemented well on a given machine architecture. Native C data types such as *int* and *float* provide the best machine access. To write a truly fast program, you have to remember to favor native types in time-critical code. In less critical code, C++ objects are useful. They provide a way to glue the program together in a manageable way. They're particularly useful in performing automated cleanup, since objects have destructors that are automatically called.

Originally in the C language, there was a requirement that all of 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. We extensively 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 will also include its own set of working variables that's 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.

Vendors of C++ compilers such as Intel and Microsoft always claim that their compiler performs miracles. The truth is that each compiler has its own strengths and weaknesses. It's useful from time to time to take a look at the compiler output. Anyone with knowledge of assembly language and a bit of curiosity could do this. It's often shocking to see how much code is generated for some constructs, and it could be quite pleasing to see just how clever the compiler is at other things.

We break our objects into two categories: those that need dynamic memory allocations and those that don't. I'll call those "fat" objects and "thin" objects. We use thin objects liberally throughout our program. We declare and use them at any point in our code and allow them to be created and destroyed frequently. The fat objects are carefully managed so that we keep them for much longer durations, often for the duration of a calculation. We also avoid any pass-by-values uses of a fat object since these perform wasteful memory allocations.

If you're not careful, it's quite easy for a C++ program to spend 20 percent or more of its execution time doing dynamic memory allocations via the new operator. This happens if lots of objects are being created and destroyed, staying around only for brief periods. We've carefully constructed our code so that no more than 1 percent of the execution time is used for dynamic memory allocations. Many objects are set up only in the initialization phase and then persist for the entire calculation. Vector classes either do no memory allocations or allocate memory initially and then carefully recycle it after that. Many of the objects we use are designed so that they don't perform dynamic memory allocations. Instead, they use stack-based allocations, which are much faster, relying on the chip itself.

Finally, let's talk about 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? We should think carefully before using the slower double-precision variables. You may need them sometimes, but probably not all of the time.

Now we'll talk about the hardware. You can optimize for different hardware. Often this involves the choice of the right compiler and the setting of the appropriate optimization switches. But sometimes compiler optimizations are dangerous and can give wrong answers, so you need to compare the release and debug builds of your software on large sets of data. Make sure they're giving the same answers.

The next big thing, of course, is 64-bit software. Because it offers improved access to large amounts of memory, we can expect some significant improvements from this source in the near future.

You should assess how the performance of software correlates to the processor used, the speed of the hard disks, the bandwidth of the network and the amount of memory. Then communicate this clearly to the users. Warn the users of things that may adversely affect the performance of your software. For example, virus checkers may have a big impact. I can't overstate the importance of this aspect of tuning for speed. That gives you the most bang for the least buck.

Let's talk about setup speed. A very important aspect of speed that's frequently overlooked is, how long does it take to set up the system to do the calculations that you want? That includes any time necessary to cover the programming of missing features and the time to map the data into the system. How long does it take from the day the software arrives until you have a fully operational model? How long does it take to modify the system to meet the changing requirements?

Our approach is radically different from that some others have taken. With a vendor-maintained system such as ours, features go into the system as fast as we can program them, but sometimes our clients need changes faster than we can manage. We had to find a way of accommodating those requirements, so we've introduced formula tables where the users can add logic, which we interpret or compile at run time. Other vendors have attacked these problems in different ways. My point here is not to argue for our way above others, but rather to emphasize the importance of this consideration. Much of the user's time will be spent setting up tables and loading seriatim or model data into the system. Try to minimize the learning curve here; the user will thank you. If you're setting something up just for one purpose, you may try a direct feed from the administration systems. If this is a multipurpose tool, then you need to offer it to all that can adjust to any data format. Either way, if you're processing large quantities of data, input speed can be significant once you include pausing, mapping and validation. This isn't the glamorous area, and it doesn't involve any special actuarial insight, but it is an important aspect.

It's my strongest wish that when processing blocks of in-force business, whether for valuation, as a liability modeling or as stochastic processing, the system you use is fast enough that you can process seriatim data without grouping or sampling. It's so much easier to get the information into the system if you're not struggling with effective grouping techniques. Can you really spend the time validating your grouping methodology? I used to be a member of the opposite camp, so I'm a born-again seriatimist.

We've talked about distributed processing already. You can use distributed processing to accelerate a modeling system by running parallel calculations on multiple processors, in one box or over multiple boxes. Ideally, if you have 20 processes, your program should run 20 times faster. The more research and development you do in this area, the closer you'll come to this ideal. It takes a lot of work to provide a simple but fully scalable model for the users. I can give you a few pointers.

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? You're only as fast as the slowest machine. How do you minimize the overhang times when some processors are busy and others are finished? You also need to think about bringing the results together from all the different processors. Speeding up the calculation is only part of the job here. 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. 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. Hardware vendors may tell you that these capabilities are built into their machines, and to a limited extent they are. But the primary load is on the software developer to build software to fully support these objectives.

I'd like to talk about grid computing. As you add more and more machines to a distributive processing farm, the complexities of managing it become greater and greater. The answer to this problem is grid computing, which automates the task of managing the farm so that many more processes can be used. As you scale up the number of processes, maintaining run time scalability becomes tougher and tougher. Even more attention must be paid to every recovery and scheduling for multiple users. In this area, it may pay to find partners with specific expertise. Companies like Platform Computing, the largest grid computing company in the world, specialize in this area, and they can help you set up your grid.

I've included a couple of charts to show the results of some experiments we ran at IBM's laboratories in San Mateo(see page 8, slides 1 and 2). On 64 machines we

were able to run about 61 times faster than on one machine. That's good scalability. The second chart shows the effect on run time of increasing the number of processes for a variety of different batch jobs. As you can see, they all scale well, but some scale much better than others. The order of the loops has the biggest impact on the ultimate scalability. You can see in this job that we reduced a 120-hour run time to two hours.

We all make mistakes. If it's a simple code error, we can fix it readily. But what if it's more basic than that? What if we have some of our fundamental architecture wrong? What if the way we chose to design the program means that the maximum speed can never be obtained? We can plow on regardless, we can go back and change the architecture or we can offer maybe two different ways to run the software—the old way and the new way.

The big deal here is admitting the mistake. The bigger the mistake, the harder it may be to admit it. For example, we developed a data management program in FoxPro separate from our model. At the time, it seemed like the right thing to do, since our model was a DOS program. Later, when we moved our model to Windows, we realized that a single application combining the model with data management would be much more flexible and efficient, but it would be a massive undertaking to get all the users over the transition. We spent several man-years in developing tools to automate that transition over and above all the work to design all the data management facilities in C++. So admitting mistakes can be very expensive, but if you don't do it, your ultimate progress may be limited. Choose short-term pain for long-term gain. Now, you won't necessarily find your mistakes unless you specifically look for them. This should be a continuous process, and you need to allocate time for it.

I can sum up by saying that speed is not something you can eke out of a system after you've developed all the functionality. Rather, you need to consider your requirements for speed up front. It can affect the programming language you use and even the equipment you target. Then you have 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. You must be aware of the compromises in functionality that you're making for speed gains, and also the converse: how much you're impairing the speed run by insisting on certain functionality or detail. Speak to the users, make sure you're on the same page or offer two or more ways to run things—a slower, more detailed method and a faster, less detailed method.

As your application develops, you need to use profiling tools to discover the roadblocks preventing your software from running at high speed. You have to be willing to invest a lot of time and effort into fine details. If you haven't done so already, now you need to research ways to introduce parallel processing into your code to take advantage of distributive processing and grid computing. Scale up your application to meet ever-increasing demands.

You should also note that many of the techniques I've covered are very labor-intensive and costly to implement. Thomas Carlisle said that genius is an infinite capacity for taking pains. I like that. By that definition, I'm a genius, and you can be one too.

MR. SABATINI: What I take away from the two sets of remarks is that to get our jobs done, particularly around modeling, we're going to need to develop the right environment, think about how we want to structure it and constantly monitor that environment to make sure that we're maximizing run time performance.

MS. JOANN WILSON: Have you ever heard of the Condor software for grid computing? I know that it's free software, and we were thinking about trying it out. Have you tried it out, and if so, can you give us any information on how it works with actuarial software?

MR. KOROGODSKY: Yes, our practice works with a number of partners, and folks from the University of Wisconsin who originated the Condor project are on the list. We've had some meetings with them. We do know that their software does work. It's free. I think it works with actuarial software, specifically with MoSes. I'll give you a word of caution, though. If you try to, for instance, implement that type of approach in the context of your company, you might have difficulties pushing that through your IT people because it's a university project. They're not a software company, they don't have proper support, and IT will raise that as flags.

There are some interesting advantages of using Condor as opposed to production grid software packages. But specifically related to what I talked about regarding the utilization of sleeping computers, Condor allows you a lot of flexibility in setting up utilization levels. For instance, you can reach out to your favorite team in the company, which is accounting, and agree with them that their utilization of a particular computer is 9 every evening to 2 every morning at 80 percent of CPU cycle. The rest you can take. Condor allows you to do things like that very flexibly. But, again, it has some drawbacks in terms of acceptability in the corporate IT environment.

MR. HUBERT MUELLER: My question to either of the panelists is on the use of sliding windows. One thing we've done with MoSes is that when you have large numbers of model points, usually our constraint is their RAM memory. Would either of you comment on the use of sliding windows and how that could help manage that better or maybe allow a larger number of model points to be projected?

MR. GOLD: I can only say that the software shouldn't limit the number of model points. In our software, you can run one model point or 100 million, and the memory requirement wouldn't change. I cannot answer your specific question.

MR. MILLER: I'm talking about how you could use that to optimize. You can run a

large number of model points in a shorter timeframe.

MR. KOROGODSKY: This question is specific to MOSES software. Others probably don't have that functionality built in. We definitely have seen that the sliding windows technique allows you to optimize the memory usage. MoSes, as well as any other second-generation actuarial modeling platform, is very memory-hungry. I think that there are a number of memory management techniques, and sliding windows is one of them. Another one is an extra 1 gigabyte of RAM. That functionality is also unique to MoSes. There are a number of good techniques, and absolutely it needs to be done and needs to be used. I think that there are issues of the MoSes modeler to which you can refer, or you can talk to us and we'll tell you how to do that.

FROM THE FLOOR: At the beginning of the session, Alex, you talked about a situation where you set up a model and were ready to run it late Friday, and then when you returned to the office Monday morning, the model had crashed a half-hour after you left on Friday. How do we handle that if we're using not in-house developed programs but using software from the market? How can we effectively set up to avoid that happening?

MR. KOROGODSKY: You do need to look into specific software packages. Again, utilization of grid vendors would help you get there because the grid engine is installed on every computer, on every CPU that you're utilizing during your run. It also has the fault tolerance monitoring functionality. What it does is report back to head node the unavailability of a certain node, and then the work gets redistributed to the other nodes that are available. It will prolong your run time obviously, because you just load a computational resource. But, again, the grid can go and seek an additional computational node that was not regionally included in yours. You might not get your run time prolonged. The bottom line is that, although you could probably spend a lifetime designing that from scratch, I would recommend that you reach out to a professional software company that deals with that and just integrate it with your homegrown solution.

MR. WILL MITCHELL: I have a question about the merits of closed code versus open code. My experience has been with a system that uses open code. One advantage I've seen is that it seems like people who are not sophisticated programmers who are working in product development can experiment with different designs. Are there some limitations with a closed code?

MR. GOLD: Closed code puts a tremendous burden on the developer of that closed code. You can't do the same amount of work as vendors of open source software and expect to get away with it because you would end up with software that was just simply not functional enough, and people would quickly move into the open source area. So you have to work a whole lot faster and a lot harder. You have bigger development teams. But the payoff is run times two orders of magnitude faster than the open code software. It's a very big payoff.

I mentioned that the solution we found to the inability of the user to get right into the code was to introduce these formula tables in our system. It was a lot of work for us to develop. But now we use this to develop our own code in Visual Basic and write VB code and have access to variables in certain places in the system. That code is simply data; it is not source code. It is compiled at run time, and it works pretty fast. But it doesn't become part of these systems, so there's no problem when there's an update. It's just data, like a premium table. It's a lot of work to put in that kind of run time compiler, but that's what you have to do to compete with open source kinds of code if you're going to have closed code.

MR. KOROGODSKY: Don't tell your IT people you're purchasing an open source system, all right? It might not fly that well. I have a couple of side comments on what Phil was saying. One is that you need to understand that when you talk about modeling environments being open code, they're open application code, that is, the model itself. The engine is still locked down by the vendor. It's a locked-down calculation engine with the application code being open. Personally, I believe that allows you a greater flexibility in performance-tuning of your particular model and your particular application, as opposed to generically the calculation engine.

MR. SABATINI: From my perspective, it's a good news/bad news situation. The open code allows you to customize the tool to do exactly what you want it to do. That's the good news. The bad news is that to do it optimally, you have to have the right kind of environment that allows you to optimize that code. I think Phil's point is that if you don't have that efficiency in the code, then by having the flexibility to change the code, you're bogging down your processing environment. To Alex's point, if you're going to work with an open code system, then you need to have a computing environment and the skill sets to make sure that you optimize that environment appropriately. Otherwise, you then pass on that responsibility to a vendor in a closed code system.

FROM THE FLOOR: We talk a lot about forecasting and using fairly new programming systems. A lot of us are probably dealing with legacy systems. How do these concepts relate to legacy systems? Can we adopt them, or do we need to go to something more modern? Also, you talked about forecasting. Can these be carried over to a production environment system as well?

MR. GOLD: There is a lot of talk about grid computing and scavenging. For production environments you're talking about—for example, it's quarter end and I have to have C-3 Phase 2 done within three days—you don't want to be relying on people's desktops. You need dedicated equipment, or something like that maybe takes place. Get through your calculations in the right amount of time. We're finding that companies are not allowing their data to be distributed across their desktops for security reasons. That's not a safe way of dealing. They don't want people running from their desktops, and they don't want people's desktops being used. We're talking dedicated machines.

MR. KOROGODSKY: In fact, it's a very good question. A trend in the last year has been, can we utilize purely modeling platforms—ALFA, MoSes, Prophet—for production environments? Can we use these environments with the controlled environments? The answer is yes. We have experience implementing systems like that. Whether you choose to take your forecasting and totally revamp it and put it in another software, or whether you choose to live with the software that you might have in house already that you've been using for the last five or 10 years, that to me is a secondary question. The question is, are you ready to step up that capability? The choice between the two, while important, is only 25 percent of the deal. You need to optimize the processes. You need to optimize the infrastructure. Even organizational change itself is going to be something to tackle. Unless you get these people, processes, technologies and strategy together to think about your planning and forecasting capability, software alone is not going to do the trick.

MR. LAWRENCE S. CARSON: My question is a follow-up to the point you just made. You talked about organizational change and starting to talk about these open application code systems. Have you found companies moving to more of a centralized modeling function or having a team of people who do nothing but optimize and sort of go back to the old system where you have a job to do and you cue it up for somebody to work on?

MR. KOROGODSKY: Definitely, yes. Everything that you say is taking place right now. We help a lot of our clients realize that dream. One of the most common approaches to that is a subgroup of a modeling team called modeling stewards. The things that I talked about (in terms of the code optimization, model architecture, working with IT people, infrastructure, hardware and all that stuff, as well as other sorts of R&D things that need to be done within the modeling unit) are the types of tasks that modeling stewards can embark on if that group exists. But, again, it's an organizational change. It's not enough to just get your senior management to buy in. Buy-in isn't going to work; you need their full engagement in driving that concept through organization. There are a lot of issues with that. It's not an easy task, but it's a task, once accomplished, that gets you to the new level absolutely.

MR. SABATINI: With open code, the whole subject of diversion control becomes extremely important. You need to establish processes to manage origin control. It doesn't matter whether it's an open code system or even a code system that allows you to make these changes; you still need to maintain that version as well.