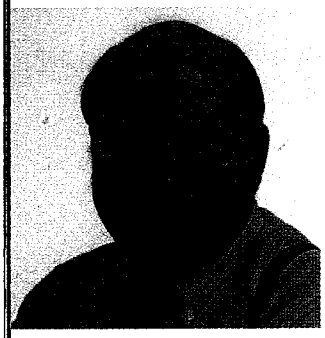
A Kilometer Is Not a Kilometer



BY ROGER SESSIONS

Buddha is quoted as saying, "A rose is not a rose. Therefore it is a rose." By that he means that until we learn to look beyond our preconceptions of a rose, we cannot see the true rose. We must look deeply and wisely at things in order to appreciate their real character. In object-oriented programming, we often become attached to our preconceptions. One of our favorite preconceptions is performance. Many people reject the whole paradigm because of "the unacceptable performance overhead." Others choose between languages based on how quickly a method can be invoked.

In this column, we are going to look more deeply (and hopefully, more wisely) at the whole issue of performance in object-oriented systems. An analogy might help to start things off. Suppose you have been assigned the task of moving x units of y a distance of z kilometers. The first thing you do is look for a vehicle in which you can load up your x units of y. Let's say you have two vehicles from which to choose. The first takes 30 seconds to travel one kilometer. The second takes three minutes. Which vehicle is best suited to your task? The answer is: You don't know. It depends on what x and y are. If x is two dozen and y is eggs, then you will choose the sports car. If x is 100 tons and y is coal, you will choose the river barge.

Performance factors

Performance is only relevant when workload is taken into account. When discussing the performance of method invocations, we need to look at three factors of workload: the cost of invoking the method, the number of times the method will be invoked, and the amount of work the method will do. The relationship between these factors is as follows:

```
TC = (CIM + COW) * NMI
  where
    TC  =  Total Cost
    CIM =  Cost of Invoking
           Method
    COW =  Cost of Work (in
           the method)
    NMI =  Number of Method
           Invocations
```

In general, when we write software, we want the Total Cost (TC) to be as low as possible. We can keep it low by focusing on any combination of the three factors that contribute to TC. Let's look at some case studies.

Case 1

In case 1, let's say the Cost of Invoking a Method (CIM) is equal to 0.80 microseconds. Keep in mind that 1,000 microseconds equals 1 millisecond and 1,000 milliseconds equals 1 second. Let's also say that the Cost of Work (COW) of that method is 1.0 microsecond. You might assume that the CIM is a significant factor in this equation, but wait! Suppose the method is only invoked once in a program that takes 10 seconds to run. The TC for that method is

```
TC = (CIM + COW) * NMI
   = (0.80 microseconds/invo-
     cation + 1.0 microsec-
     onds/invocation) *
     1 invocation
   = (0.80 + 1.0) * 1
   = 1.80 microseconds
```

According to these calculations, our method only contributes 1.80 microseconds out of a total run time of 10 seconds (10 million microseconds). The total fraction of time spent in this method is 1.80/10,000,000—an infinitesimal portion. People worrying about the performance of this method should be worrying more about their own performance!

Case 2

Let's look again at case 1, but this time, let's ask a different question: How many times would we have to invoke the method before the TC of that method would equal at least 1% of the run time of the whole program? The run time of the whole program is 10 seconds, or 10 million microseconds. We will reach 1% of 10 million microseconds when we reach 100,000 microseconds. Plugging these new numbers into the equation yields

```
TC       = (CIM + COW) * NMI

100,000 microseconds =
           (0.80 microseconds/
           invocation +
           1.0 microseconds/
           invocation) *
           x invocations
100,000  = (0.8 + 1.0) * x
         = 1.8 x

Therefore,
x        = 100,000/1.8
         = 55,555
```

In other words, we have to invoke this method more than 55,000 times before it contributes even 1% to the run time of this program.

Relative costs of invoking methods

What is the cost of invoking a method? The answer to this question depends both on the type of method and what we are using as a base line. Let's look at the cost of running a given code segment using each of five different coding techniques. Each code technique offers both increased benefits and increased costs. In this section, we'll see how these two are related.

When studying the measurements I give, you should focus more on the relationships between numbers and the measurement techniques rather than on the absolute numbers, which will change depending on compiler optimization, machine configuration, and environment conditions. My particular machine is a 100MHz processor with 16MB of memory, and I have not used any code-optimization compiler switches.

Let's start by looking at standard C++ virtual method invocation. My C++ dog definition and implementation is shown in Listing 1.

As you can see from Listing 1, my dog supports a method named **runOneKilometer**. I am going to measure the cost of invoking this method.

My measurement program is shown in Listing 2.

As you can see from Listing 2, I can run this program with various numbers of iterations. I usually run the program at least three times choosing iteration numbers resulting in run times between 10 and 240 seconds. Less than that makes for inaccurate measurements. More than that makes for a high boredom factor. I start my measurements as soon as the iteration starts and stop when the iteration stops.

I ran this program with 20, 50, and 100 million iterations, yielding run times of 16, 43, and 81 seconds, respectively. The program automatically calculates the milliseconds per kilometer, which for my runs came out to be .00080, .00086, and .00081, respectively. Let's use the average: .00082 milliseconds.

Next, let's look at our base line, which does the equivalent work without using object-oriented programming.

I have used two baselines. The first (Listing 3) does the equivalent work completely inline. The second (Listing 4) does the equivalent work in a procedure call.

I ran Listing 3 (the inline program) with 100, 200, and 500 million iterations, yielding run times of 19, 38, and 94 seconds, and all weighing in at the identical .00019 milliseconds per iteration.

I ran Listing 4 (the procedural version) with 50, 100, and 200 million iterations, yielding run times of 29, 57, and 113 seconds, weighing in at .00056 ± .00001 milliseconds per iteration

The difference between using C++ virtual methods (.00082 milliseconds) and procedure calls (.00056 milliseconds) is .00082 – .00056, or .00026 milliseconds. The

Listing 1: C++ dog.

Definition:

```
class dog {
  public:
    virtual void runOneKilometer();
    long int howManyKilometers();
    dog();
  private:
    long int kilometersRan;
};
```

Implementation:

```
#include "dog.hpp"

void dog::runOneKilometer()
{
  kilometersRan++;
}
dog::dog()
{
  kilometersRan=0;
}
long int dog::howManyKilometers()
{
  return kilometersRan;
}
```

Listing 2: C++ Measurement program.

```
#include <stdio.h>
#include <stdlib.h>
#include "dog4.hpp"

int main()
{
  long iterations;
  long n, seconds;
  float mspk;
  dog4 *snoopy = new dog4();

  printf("Measuring virtual C++ method invocations.\n");
  printf("How many kilometers? ");
  fflush(stdout);
  scanf("%d", &iterations);

  for (n=0; n<iterations; n++) {
    snoopy->runOneKilometer();
  };

  printf("Snoopy ran %d kilometers\n",
     snoopy->howManyKilometers());
  printf("How many seconds did that take? ");
  fflush(stdout);
  scanf("%d", &seconds);
  mspk = (float) (seconds * 1000) / (float) (iterations);
  printf("Total milliseconds to go one kilometer: %10.6f", mspk);
  exit(0);
}
```

difference between using procedure calls (.00056 milliseconds) and inline code (.00019 milliseconds) is .00056 – .00019, or .00037 milliseconds.

In other words, it is significantly more expensive to go from inline coding to procedural coding than it is to go from procedural coding to objects. Yet, the wisdom of procedures is universally accepted, while many wonder if they can "afford the penalty" of using objects.

This is a very important point and cannot be overemphasized. The use of object-oriented programming does not result in any significant performance degradation. Please do whatever you can to dispell this myth.

Let's go higher up the food chain. Standard SOM objects add language independence and upward binary compatibility to C++ objects. DSOM adds the ability to distribute objects across machines. What do we pay for these benefits?

Listing 5 shows a SOM equivalent of the C++ dog in Listing 1.

This dog can be instantiated locally or remotely. Listing 6 shows a local instantiation; Listing 7 shows a remote instantiation. Listing 7 is substantially like Listing 6, so I have shown only the significant difference, which is in the object instantiation.

The run time for Listing 6, which adds standard SOM benefits to the C++ object, is 13, 24, and 59 seconds for 10, 20, and 50 million invocations, weighing in at .0012 milliseconds. The difference between SOM and C++ (about .00038 miliseconds) is similar to the difference between C++ and procedures (.00026 milliseconds).

Let's put this in context of some actual work. I modified Listing 2 to measure the time required to fill a 100-byte buffer with a given character. Most of the code is unaffected. However, the new loop is shown in Listing 8.

The program shown in Listing 8 took .019 milliseconds/iteration to run. How would the cost of running one iteration of this code as a C++ object compare to the cost of running one iteration of this code as a SOM local object? Let's go back to our formula:

```
TC = (CIM + COW) * NMI

where
  TC  =  Total Cost
  CIM =  Cost of Invoking
         Method
  COW =  Cost of Work (in
         the method)
  NMI =  Number of Method
         Invocations
```

For C++, the cost for one iteration would be:

```
TC = (.00083 + .019) milli-
     seconds/iteration
   = .01983
```

For SOM, the cost for one iteration would be:

```
TC = (.0012 + .019) milli-
     seconds/iteration
   = .0202
```

The difference between SOM (local) and C++ both doing this relatively modest amount of work is less than 2%.

What about distributing the object using DSOM? Here I would expect to see a significant decline because of

Listing 3: Inline program.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  long iterations;
  long kilometers = 0;
  long n, seconds;
  float mspk;

  printf("Measuring in line
      code.\n");
  printf(
      "How many kilometers? ");
  fflush(stdout);
  scanf("%d", &iterations);

  for (n=0; n<iterations; n++) {
     kilometers++;
  };

  printf("How many
      seconds did that take? ");
  fflush(stdout);
  scanf("%d", &seconds);
  mspk = (float) (
      seconds * 1000) / (
      float) (kilometers);
  printf("Total milliseconds to
      go one kilometer: %10.6f",
      mspk);
  exit(0);
}
```

Listing 4: Procedural program.

```
#include <stdio.h>
#include <stdlib.h>
void runOneKilometer(long *value);

int main()
{
  long iterations;
  long kilometers = 0;
  long n, seconds;
  float mspk;

  printf("Measuring procedure call.\n");
  printf("How many kilometers? ");
  fflush(stdout);
  scanf("%d", &iterations);

  for (n=0; n<iterations; n++) {
     runOneKilometer(&kilometers);
  };

  printf("How many seconds did that take? ");
  fflush(stdout);
  scanf("%d", &seconds);
  mspk = (float) (seconds * 1000) / (float) (kilometers);
  printf("Total milliseconds to go one kilometer: %10.6f",
      mspk);
  exit(0);
}
void runOneKilometer(long *value)
{
  (*value)++;
}
```

the work involved in marshalling method calls. (If unfamiliar with DSOM, see the April 1996 issue, "Distributed Objects," pp. 44-47.)

I ran Listing 7 in the best possible case, on a single machine with no overhead for networking. I was using the beta version of SOM 3.0.

I ran 10, 20, and 30 thousand iterations. I had to go from millions of iterations to thousands to complete the tests in a reasonable time. The iteration run times were 66, 131, and 199 seconds, for an average iteration time of 6.59 milliseconds.

This average iteration time was almost a 5,500-fold decline from local SOM. In other words, you could run almost 5,500 local SOM methods for the cost of one remote DSOM method—which may seem like a huge degradation, but what are we getting for this cost? Object Distribution!

Comparing DSOM to C++ virtual methods doesn't make sense. We would never use DSOM for small C++–like classes. A fair comparison is DSOM to Remote Procedure Calls (RPC), the procedural equivalent to a remote method call. Here DSOM compares very favorably.

The cost of running DSOM across a network is approximately twice the cost of running it on a single machine, bringing the cost to about 14 milliseconds, about 30% more expensive than running RPC (RPC performance numbers provided by Virgil Albaugh). This cost difference is similar to the cost difference between procedures and virtual methods. Keep in mind that DSOM 3.0 is still in beta and will probably improve.

In order to use DSOM effectively, you need to make sure that the cost of running the method is at least 10 times the cost of invoking the method. This means that DSOM should not be used for method workloads of less than about 140 milliseconds.

On my system, 70 milliseconds is approximately the time required to open a file, write a single element, and close the file. For this workload, DSOM will contribute only about 20% of the total overhead of the method invocation, about the same as RPC would have contributed.

There is another scenerio under which DSOM makes sense. If the number of method calls is low, the

Listing 5: SOM dog.

Definition:

```
#include <somobj.idl>
interface somdog : SOMObject {
  void runOneKilometer ();
  long howManyKilometers();
  implementation {
    long kilometersRan;
    override: somDefaultInit;
    somDefaultInit: init;
    releaseorder: runOneKilometer, howManyKilometers;
    dllname = "dog.dll";
  };
};
```

cont'd on p. 46

Listing 6: Local instantiation of SOMDog #include <stdio.h>.

```
#include <stdlib.h> #include "somdog.h"

int main()
{
  long iterations;
  long n, seconds;
  float mspk;
  Environment *ev;
  somdog snoopy = somdogNew();

  ev = SOM_CreateLocalEnvironment();

  printf("Measuring language independent SOM method
      invocations.\n");
  printf("How many kilometers? ");
  fflush(stdout);
  scanf("%d", &iterations);
  for (n=0; n<iterations; n++) {
    _runOneKilometer(snoopy, ev);
  };
  printf("Snoopy ran %d kilometers\n", _
      howManyKilometers(snoopy, ev));
  printf("How many seconds did that take? ");
  fflush(stdout);
  scanf("%d", &seconds);
  mspk = (float) (seconds * 1000) / (float) (iterations);
  printf("Total milliseconds to go one kilometer: %10.6f", mspk);
  exit(0);
```

Listing 7: Remote instantiation of SOMDog.

```
/* ... */
  ev = SOM_CreateLocalEnvironment();
  SOMD_Init(ev);

  snoopy = _somdNewObject(SOMD_ObjectMgr, ev, "somdog", "");

/* ... */
```

Listing 8: Loop with buffer filling code.

```
  for (n=0; n<iterations; n++) {
     kilometers++;
     for (n2=0;n2<200 ;n2++) {
       buff1[n2] = 'a';
     }
     buff1[200] = 0;
  };
```

Need a Software Product? Find it on the Web!
http://www.mfi.com/softwareguide/
AIX, OS/2® tools and more!

Listing 5: *Cont'd from 45.*

```
Implementation:
#include "somdog.h"

SOM_Scope void SOMLINK runOneKilometer(somdog somSelf,
        Environment *ev)
{
    somdogData *somThis = somdogGetData(somSelf);
    somdogMethodDebug("somdog","runOneKilometer");
    _kilometersRan++;
}
SOM_Scope long SOMLINK howManyKilometers(somdog somSelf,
        Environment *ev)
{
    somdogData *somThis = somdogGetData(somSelf);
    somdogMethodDebug("somdog","howManyKilometers");
    return _kilometersRan;
}
SOM_Scope void SOMLINK somDefaultInit(somdog somSelf,
        som3InitCtrl* ctrl)
{
    somdogData *somThis; /* set in BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    somdogMethodDebug("somdog","somDefaultInit");
    somdog_BeginInitializer_somDefaultInit;
    somdog_Init_SOMObject_somDefaultInit(somSelf, ctrl);
    _kilometersRan=0;
}
```

overall cost of DSOM will also be low. For example, 10 DSOM method calls in a program run time of 10 seconds will contribute about 140 milliseconds—about 1% of the total run time. These numbers, however, do not include the cost of "finding" remote objects, which is very significant in DSOM, but which I am assuming is part of the start-up cost of the program.

So we have two choices with DSOM. We can either use it for expensive method calls, or we can use it sparingly. Either gives us the benefits of Distributed Object Programming with reasonable cost. And regardless of how we use DSOM, its cost is comparable to other distribution techniques such as RPC.

Now, those that expect to take existing C++ classes, rewrite them with DSOM, and have all their classes magically distributed, are in for a nasty shock. Object distribution requires careful planning at the front end of a project. Not a bad place to spend some of your consulting dollars, if I may be forgiven for such a self-serving point of view.

Epilogue

Once again, keep in mind that the absolute numbers I have reported here are much less important than the relationships between the numbers. There is no substitute for running your own performance tests.

To help you in this endeavor, all of the code in this article is available from the SOMObjects Home Page, located at *http://www.fc.net/~roger/owatch.htm*. Look for a link leading you to code for *OS/2 Magazine* articles.

Always remember that the cost of a method is related to the cost of the method invocation, the workload of the method, the number of times the method will be invoked, and the overall run time of the program. A method is not a method. A kilometer is not a kilometer. A rose is not a rose. OS/2

Roger Sessions is president of ObjectWatch Inc., a company specializing in training and consulting in the use of SOM, DSOM, and related object-oriented technologies. He has spoken at over 30 conferences and has written extensively. His three books include the newly published Object Persistence: Beyond Object-Oriented Databases. *Roger also publishes the SOMobjects Home Page (http://www.fc.net/~roger/owatch.htm) and an Internet newsletter called* ObjectWatch on SOM. *He can be contacted via e-mail at roger@fc.net.*