overload 110

AUGUST 2012 £3

Large Objects and Iterator Blocks

We see how iterator blocks can help to avoid .Net memory issues

Replace User, Strike Any Key?

Is the user *really* the primary source of all problems in IT?

Black-Scholes In Hardware

We use a financial model to illustrate how to implement hardware algorithms

Valgrind: Advanced Memcheck

Investigating the more advanced Valgrind facilities for diagnosing memory errors

Simple Mock Objects for C++11

We see how to use the new language features provided by C++11 to implement mock objects in our unit tests

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 110

August 2012

ISSN 1354-3172

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines

All articles intended for publication in Overload 111 should be submitted by 1st September 2012 and for Overload 112 by 1st November 2012.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Allow Me To Introduce Myself

Using data mining techniques to write an editorial.

In *Overload* 108, Ric Parkin said goodbye as *Overload* editor after a four year stint. Allow me to introduce myself. I'm Fran Buontempo and I am your new editor. Choosing a suitable topic for editorials is difficult, yet it seems suitable to use this as an opportunity to reminisce about issues gone by, as we say goodbye to our old editor, while obviously looking forward to many articles from him in the future. Many months ago, in January in fact, Nigel Lister posted a word cloud of this year's conference on accu-general [Lister12]. This is a beautiful way of representing the frequency of words contained in documents. A more traditional approach would present a histogram, with bars showing how many times an item appears. Wikipedia [Histogram] suggests these were invented by Karl Pearson, though I like to think the ideas trace back to Florence Nightingale's innovations in statistical graphics, such as the rose diagrams [Nightingale]. I therefore produced a word cloud of *Overload* 108 [Overload], and was pleased to see the words 'Surreal' 'Mutation' standing out proudly.

Word clouds are part of the growing 'Big data' trend, which seems to be one of the latest buzz-words [Gigaom]. Though big data involves the hardware to deal with vast quantities of bits and bytes, at its heart is the attempt to extract information from data, which can be used to make money through smart business decisions, to cure cancer or to categorise proteins or new galaxies. I regard big data as the trendy face of data mining and machine learning. These disciplines are related to statistics, though encompass a much broader scope of approaches including swarm-inspired algorithms such as ant-colony optimisations, other nature inspired approaches such as neural networks and genetic algorithms, as well as clustering and classification and many other ways of searching data for meaning, or at least patterns. On a smaller scale, data mining and machine learning can provide a way to reflect and reminisce on historical trends, for example, issues of a magazine. Communications, the ACM members magazine, recently ran an article using n-Grams to analyse its previous content [ACM]. The motivation of the article was to delve into the institution's identity, considering its worldwide readership, long history and churn of members, using previous publications as input. As an organisation, the ACCU seems to have been through a time of similar reflection, for example musing on the 'Professionalism in Programming' tag line on accu-general. The coincidence between the ACM musings and our search for identity, and amused by 'surreal' 'mutations' in the *Overload* 108 tag cloud, the most sensible option for my first editorial had to be to get a computer to write it for me. I'm a geek, so what did you expect?

By saving all the words in *Overload* 103–108 inclusive in separate text files, and applying the Porter stemmer algorithm, [Porter], a tally chart of word frequencies for each edition can be produced. This algorithm trims or stems words such as 'mutation' and 'mutated' to 'mutat', so they are counted as the same word. When this is applied to several journals the information can be combined to graph the top *n* words for each issue, over time. Taking care to insert zeros for runs where words disappear off the radar, this can be used to look for trends. Using so few articles will almost certainly not reveal any long term trends, but will hopefully give a starting point for further investigation. Table 1 shows the frequencies of the top four stem words over the articles considered, and Figure 1 graphs this for us. Immediately 'test' jumps out as the highest scorer in two different issues, by a large margin. Perhaps this is a topic that captures our imagination at periodic intervals. Next, certain words seem to have a mini-trend for two or three articles running such as 'function', 'code', 'list' and 'type'. The stemmer algorithm will chop short words, such as C++, C, Go, R, Q, so it might be interesting to

Word	Overload Editions					
	103	104	105	106	107	108
function	0	0	0	112	100	0
differ	0	0	67	0	0	0
code	0	74	109	0	86	148
develop	0	78	0	0	0	0
specif	107	0	0	0	0	0
express	0	0	0	0	158	0
list	0	0	0	91	134	0
except	145	0	0	0	0	0
equal	135	0	0	0	0	0
unit	0	0	0	0	0	138
channel	0	0	0	91	0	0
file	0	86	0	0	0	0
error	0	0	62	0	0	0
test	0	308	0	0	0	309
mutat	0	0	0	0	0	158
type	0	0	80	145	0	0
valu	109	0	0	0	0	0

Table 1

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

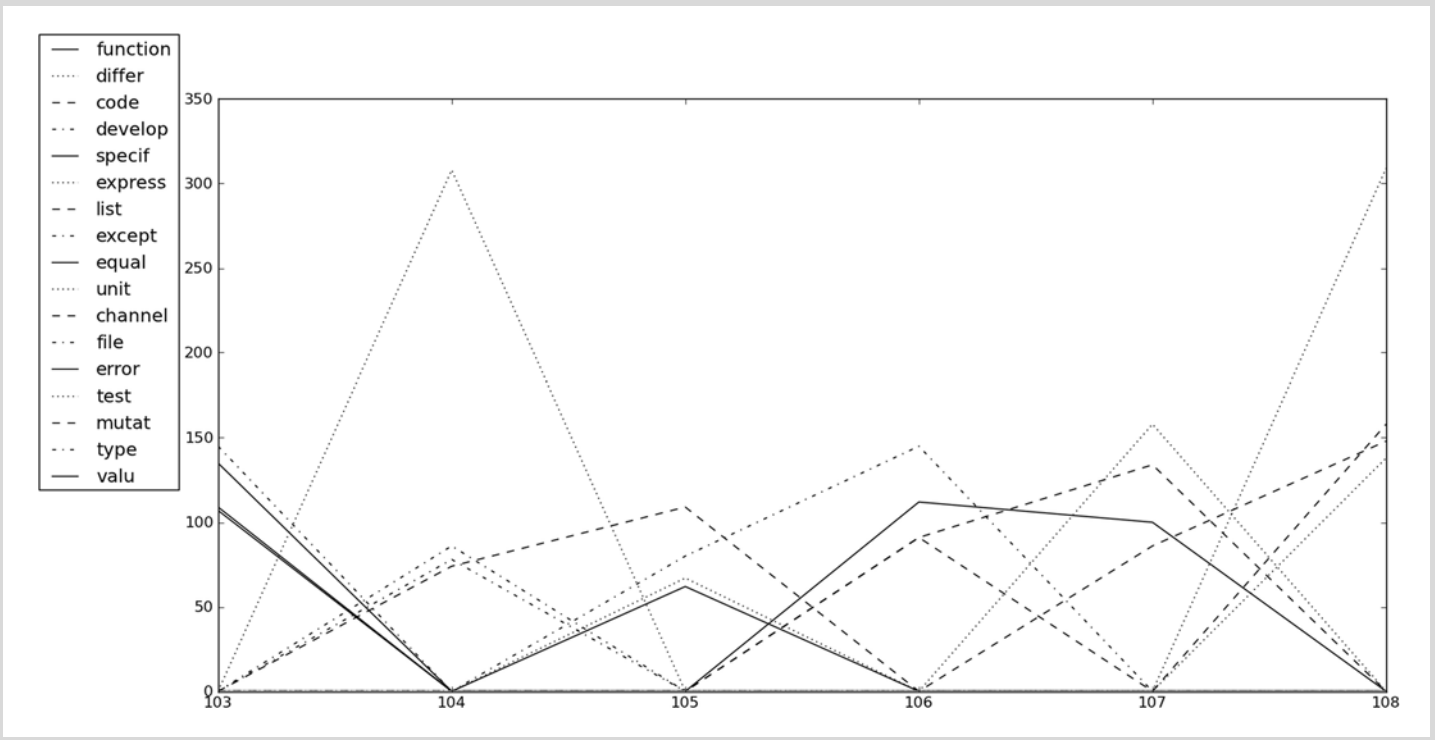


Figure 1

adapt it to include language specific words that it would otherwise filter out, taking care to disambiguate 'go' and the language 'Go'. Increasing the number of top words considered will clearly reveal further trends and mini-trends, but filling my editorial with computer generated graphs and tables might be considered cheating.

I look forward to the future editions of Overload, and would like to thank Ric for all his hard work, including keeping an eye on my first issue as editor. Thanks also to the Overload review team, and welcome to Chris Oldwood who has just come on board.

References

[ACM] Communications of the ACM, Vol 55, No 5, 2012

[Gigaom] http://gigaom.com/cloud/big-data-the-quick-and-the-dead/

[Histogram] http://en.wikipedia.org/wiki/Histogram

[Lister12] http://dl.dropbox.com/u/34106607/ACCU_SCHEDULE_smaller.png

[Nightingale] http://en.wikipedia.org/wiki/Florence_Nightingale

[Overload] http://www.wordle.net/show/wrdl/5128616/overload108

[Porter] http://www.tartarus.org/~martin/PorterStemmer

Valgrind Part 3 Advanced memcheck

Valgrind provides several mechanisms to locate memory problems in your code. Paul Floyd shows us how to use them.

In the previous part of this series I covered basic use of memcheck. In this article, I'll expand on that and cover the difficult cases that I touched on previously:

1. Compiling with Valgrind macros.
2. Attaching a debugger.
3. Using memory pools.

Compiling with Valgrind macros

When you are testing an application with memcheck, you have a passive role interacting with Valgrind. Valgrind will only generate output if an error occurs (not counting the header that contains copyright information, the Valgrind options in effect, shared libraries and function intercepted and the footer with a summary of errors found and suppressions used). You have no access to the internals of the VEX virtual machine or the state of memory. Valgrind provides you with macros that allow you to actively control output and interact with the VM.

In order to trace down the precise origins of an error, you might want to generate output at points prior to the error. Alternatively, you might want to examine memory even when there are no errors. You can think of the macros for this purpose as being a bit like printf statements, with the output going into Valgrind's output (the console or the log file). In addition to Valgrind's output, the macros may return a value, either 'directly' from the macro as a status, or through inout arguments to the macro. There are also macros to trigger Valgrind actions like performing a leak check (which otherwise will only happen when the application under test terminates).

In your C or C++ source file, you have to include the appropriate header, e.g.,

```
#include "valgrind/memcheck.h"
```

(you might prefer to use `<memcheck.h>` if Valgrind is installed with its include files in the system header directories).

Then you need to add the include path to the compiler directive, if the headers are not in the system include path. For instance, in a GNU makefile

```
CPPFLAGS += -I "/Applications/valgrind/include"
```

Then you can use the macros in your source. Since Valgrind does not link any extra libraries, these macros use a different mechanism. The macros contain a sequence of machine instructions that no known compiler would ever issue and that have no side effects. The Valgrind virtual machine detects this sequence and instigates a client request. When not running under Valgrind, there is no effect other than a very small time penalty. For example, on x86 the following is used:

Paul Floyd has been writing software, mostly in C++ and C, for over 20 years. He lives near Grenoble, on the edge of the French Alps, and works for Mentor Graphics developing a mixed signal circuit simulator. He can be contacted at pjfloyd@wanadoo.fr.

```
VALGRIND_MAKE_MEM_NOACCESS(_qzz_addr,_qzz_len)
VALGRIND_MAKE_MEM_UNDEFINED(_qzz_addr,_qzz_len)
VALGRIND_MAKE_MEM_DEFINED(_qzz_addr,_qzz_len)
VALGRIND_MAKE_MEM_DEFINED_IF_ADDRESSABLE(_qzz_add
r,_qzz_len)
VALGRIND_CREATE_BLOCK(_qzz_addr,_qzz_len,
_qzz_desc)
VALGRIND_DISCARD(_qzz_blkindex)
VALGRIND_CHECK_MEM_IS_ADDRESSABLE(_qzz_addr,_qzz_
len)
VALGRIND_CHECK_MEM_IS_DEFINED(_qzz_addr,_qzz_len)
VALGRIND_CHECK_VALUE_IS_DEFINED(__lvalue)
VALGRIND_DO_LEAK_CHECK
VALGRIND_DO_ADDED_LEAK_CHECK
VALGRIND_DO_CHANGED_LEAK_CHECK
VALGRIND_DO_QUICK_LEAK_CHECK
VALGRIND_COUNT_LEAKS(leaked, dubious, reachable,
suppressed)
VALGRIND_COUNT_LEAK_BLOCKS(leaked, dubious,
reachable, suppressed)
VALGRIND_GET_VBITS(zza,zzvbits,zznbytes)
VALGRIND_SET_VBITS(zza,zzvbits,zznbytes)
```

Listing 1

```
#define __SPECIAL_INSTRUCTION_PREAMBLE        \
  "roll $3,  %%edi ; roll $13, %%edi\n\t"     \
  "roll $29, %%edi ; roll $19, %%edi\n\t"
```

which rotates EDI by 64bits, leaving it unchanged.

There are numerous such macros: Listing 1 shows the client macros in `memcheck.h`.

Let's take a look at an example (Listing 2).

This is intended to be built on a 64bit system, though the results should be similar on a 32bit system.

A pointer to **`int`**, **`pi`**, gets assigned to 2 **`int`**s worth (8 bytes) in the heap. The first **`int`** is initialized. Then I do some nasty casting, first to initialize the first half (2 bytes) of the second **`int`**. Then, with recourse to a **`struct`** with a bitfield, I initialize just two bits in the last byte of the 2nd **`int`**. So of the 4 bytes in that 2nd **`int`**, the 1st two are initialized, the third is uninitialized and the fourth has 2 bits initialized.

After all of the initialization (or not) come Valgrind client request macros. The first checks if the 8 bytes allocated are addressable. The second checks if 9 bytes are addressable. The third gets the initialization status of each of the bits that were allocated.

If I compile this and run it outside of Valgrind I get Listing 3. However, running it under Valgrind gives Listing 4.

As expected, the check whether the 8 bytes were addressable returns 0, meaning that they are all addressable. The check whether 9 bytes are accessible provokes a 'Unaddressable byte(s) found during client check request' message with information and a return of the address of the first

```
//  main.c
//  clientreq
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "memcheck.h"

struct two_bit
{
  char foo:2;
  char :0;
};
int main (int argc, const char * argv[])
{
  const size_t size = 2*sizeof(int);
  int *pi = malloc(size);
  short *ps;
  struct two_bit *ptb;

  pi[0] = 1;
  ps = (short *)pi;
  ps[2] = 2;
  ptb = (struct two_bit *)pi;
  ptb[7].foo = 3;

  unsigned long addressable
     = VALGRIND_CHECK_MEM_IS_ADDRESSABLE
       (pi, size);
  printf("addressable %lx\n", addressable);
  addressable
     = VALGRIND_CHECK_MEM_IS_ADDRESSABLE
       (pi, size+1);
  printf("addressable %lx\n", addressable);
  int status = 0;
  unsigned char bits[8];
  memset(bits, 0, 8);
  status = VALGRIND_GET_VBITS(pi,bits,size);
  for (int i = 0; i < size; ++i)
  {
    printf("byte %d bits %x\n",
           i, (unsigned int)bits[i]);
  }
  free(pi);
  return 0;
}
```

Listing 2

unaddressable byte. The loop over the 8 bytes that were allocated show that the 1st 6 bytes have been initialized, byte 6 is uninitialized and byte 7 has the bottom 2 bits initialized and the top 6 bits uninitialized.

So at the cost of having to change how the executable was built, we've gained access down to the bit of the memory status of the executable. That's great, but it does have the drawback of being static – you can't easily change at runtime what is analysed. Since Valgrind 3.7.0, there is a way to have more dynamic access to the internals while the executable is running, and that is to use the built in gdbserver. Not only can you access

```
  addressable 0
  addressable 0
  byte 0 bits 0
  byte 1 bits 0
  byte 2 bits 0
  byte 3 bits 0
  byte 4 bits 0
  byte 5 bits 0
  byte 6 bits 0
  byte 7 bits 0
```

Listing 3

```
addressable 0
==3089== Unaddressable byte(s) found during client
check request
==3089==    at 0x100000D46: main (in /Users/paulf/
Library/Developer/Xcode/DerivedData/clientreq-
einugynxilcucqauaotevhsuanfx/Build/Products/
Debug/clientreq)
==3089==  Address 0x1000040e8 is 0 bytes after a
block of size 8 alloc'd
==3089==    at 0xD6D9: malloc
(vg_replace_malloc.c:266)
==3089==    by 0x100000C3E: main (in /Users/paulf/
Library/Developer/Xcode/DerivedData/clientreq-
einugynxilcucqauaotevhsuanfx/Build/Products/
Debug/clientreq)
==3089==
addressable 1000040e8
byte 0 bits 0
byte 1 bits 0
byte 2 bits 0
byte 3 bits 0
byte 4 bits 0
byte 5 bits 0
byte 6 bits ff
byte 7 bits fc
```

Listing 4

information like that shown above, you can also (almost) debug the application like a real application directly under gdb.

Let's see an example of using the gdbserver. First of all, some example code, with a `print` function that reads beyond the array that is passed to it (Listing 5).

If I compile and run it, I get 'element 0 0' to 'element 11 0'. Running it under valgrind with the `-v` option causes the following to be included in the output (Listing 6).

I was using xterms to do this, and if you are using terminals, either you need to be very good at coping with the spliced gdb/application under test

```
#include <iostream>
#include <unistd.h>
using std::cout;

template<typename T>
void init(size_t size, T* ptr)
{
  for (size_t i = 0; i < size; ++i)
  {
    ptr[i] = 0;
  }
}
template<typename T>
void print(size_t size, T* ptr)
{
  for (size_t i = 0; i < size; ++i)
  {
    cout << "element " << i << " " << ptr[i]
         << "\n";
  }
}
int main()
{
  //sleep(10);
  int *pi = new int[11];
  init(10, pi);
  print(11, pi);
  delete [] pi;
}
```

Listing 5

```
==12922== TO DEBUG THIS PROCESS USING GDB: start
GDB like this
==12922==   /path/to/gdb ./vg_gdb
==12922== and then give GDB the following command
==12922==   target remote | /usr/lib/valgrind/../
../bin/vgdb --pid=12922
==12922== --pid is optional if only one valgrind
process is running
```

Listing 6

input and output, or you just use two terminals, which is what I did. The 'sleep' was uncommented to give a bit of time to attach gdb. In the first terminal,

```
gdb ./vg_gdb
```

(to be ready with the gdb prompt)

then in the second terminal

```
valgrind -v ./vg_gdb
```

Select the text and then quickly switch back to the first terminal and paste

```
(gdb) target remote | /usr/lib/valgrind/../../
bin/vgdb –pid=12922
```

Then I could use all of the usual gdb commands like n(ext), s(tep), p(rint) and so forth. I stepped as far as the print function.

In order to examine **ptr** I issued the command

```
(gdb) monitor get_vbits 0x59ff040 44
```

and got back

```
00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000
00000000 00000000 ffffffff
```

monitor is the command that gdb uses to communicate with a remote server. You should only use it in cases like this and not when you are debugging an application directly.

As expected, the last **int** is not initialized, as shown by the **f**s.

You don't have to use terminals, this will also work with GUI applications and GUI wrappers for gdb (like ddd).

You can use the gdbserver for several things as well as **get_vbits**.

- Information about errors that have been detected.
- Changing logging options.
- Change the accessibility flags for given memory.
- Check that memory is addressable.
- Check for leaks.

See **monitor help** for details.

If you are using Valgrind prior to 3.7.0, then you will not have this feature available. You'll need to use either or both of the macros (as described above) and the **--db-attach=yes** option. With this option set, when memcheck encounters an error, it will ask you if you want to attach a debugger, like this:

```
==9654== ---- Attach to debugger ? ---
[Return/N/n/Y/y/C/c] ----
```

If you type **y** or **Y** it will launch gbd and attach it to the application under test. With the attached debugger you have a static image of the application – you can do things like go up and down the stack and examine variables, but you can't step or run the application. When you quit gdb, control returns to memcheck running the application under test. Valgrind defaults to using gdb. You can specify another debugger with the command:

```
–db-command=<command>
```

I've had trouble with this when I've used it in the `.valgrindrc` file. When the commands are parsed, they are split on spaces, and this option usually contains spaces and **%f** for the application file and **%p** for the pid. So if my `.valgrindrc` contains

```
--db-command="ddd %f %p"
```

```
#include <iostream>
class MemPool
{
public:
  MemPool();
  ~MemPool();
  int *allocInt();
  void freeInt(int *ptr);
private:
  int *pool;
  unsigned int freeMap;
  static const size_t poolSize = 32;
};
MemPool::MemPool() : pool(new int[poolSize]),
                             freeMap(0U)
{
}
MemPool::~MemPool()
{
  delete [] pool;
}
int *MemPool::allocInt()
{
  for (size_t i = 0; i < poolSize; ++i)
  {
    if (!(freeMap & 1U << i))
      {
        freeMap |= 1 << i;
        return &pool[i];
      }
  }
  return 0;
}
void MemPool::freeInt(int *ptr)
{
  for (size_t i = 0; i < poolSize; ++i)
  {
    if (ptr == &pool[i])
      {
        freeMap &= ~(1 << i);
        return;
      }
  }
}
int main (int argc, const char * argv[])
{
  MemPool mempool;
  int *ptrs[3];
  ptrs[0] = mempool.allocInt();
  ptrs[1] = mempool.allocInt();
  ptrs[2] = mempool.allocInt();
  mempool.freeInt(ptrs[0]);
  mempool.freeInt(ptrs[2]);
}
```

Listing 7

and I run

```
valgrind --db-attach=yes xemacs
```

then I get

```
valgrind: Bad option: %f
```

This will work if you put the commands on the command line

```
valgrind --db-attach=yes --db-command="ddd %f %p"
xemacs
```

Another thing that can be difficult is if you use gdb with a command line application. In this case the output of the application will be mixed with the output (and input) of gdb.

For the last section in this article, I'll look at using memory pools. Let's start with a little noddy application with memory pool (Listing 7).

```
==9886== HEAP SUMMARY:
==9886==     in use at exit: 0 bytes in 0 blocks
==9886==   total heap usage: 1 allocs, 1 frees,
128 bytes allocated
==9886==
==9886== All heap blocks were freed -- no leaks
are possible
==9886==
==9886== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 4 from 4)
```

Listing 8

Note the obvious 'leak', 3 calls to `allocInt` but only 2 calls to `freeInt`. Compiling and running this with memcheck detects no errors (Listing 8).

Let's now add the Valgrind machinery to instrument the memory pool. The parts that need to be changed are:

1. The constructor, to tell Valgrind about the memory pool and to mark it as 'noaccess'.
2. The destructor, to actually perform the leak checks and to tell Valgrind that the memory pool is no longer used.
3. The allocator, so that Valgrind knows when a chunk in the pool is used.
4. The deallocator, so that Valgrind knows when chunks in the pool are released (Listing 9).

Note that the memory 'leaked' from the pool is marked as 'still reachable' rather than as one of the 'lost' categories.

That wraps it up for memcheck. Before I go, a few production notes. On my Mac (with Mac OS X 10.6.8 on an Intel CPU) I couldn't get the memory pool example to work. On the Linux install that I used for the same example (openSUSE 11.4) the Valgrind headers were missing and I had to add the valgrind-devel package.

In my next article, I'll cover Callgrind, a tool for time profiling applications. ■

```
#include <iostream>
#include "valgrind/memcheck.h"

class MemPool
{
public:
  MemPool();
  ~MemPool();
  int *allocInt();
  void freeInt(int *ptr);
private:
  int *pool;
  unsigned int freeMap;
  static const size_t poolSize = 32;
};

MemPool::MemPool() : pool(new int[poolSize]),
   freeMap(0U)
{
  VALGRIND_MAKE_MEM_NOACCESS(pool,
     poolSize*sizeof(int));
  VALGRIND_CREATE_MEMPOOL(pool,
     poolSize*sizeof(int), 0);
}

MemPool::~MemPool()
{
  VALGRIND_DO_LEAK_CHECK;
  VALGRIND_DESTROY_MEMPOOL(pool);
  delete [] pool;
}
```

Listing 9

```
int *MemPool::allocInt()
{
  for (size_t i = 0; i < poolSize; ++i)
  {
    if (!(freeMap & 1U << i))
    {
      freeMap |= 1 << i;
      VALGRIND_MEMPOOL_ALLOC(pool, &pool[i],
      sizeof(int));
      return &pool[i];
    }
  }
  return 0;
}

void MemPool::freeInt(int *ptr)
{
  for (size_t i = 0; i < poolSize; ++i)
  {
    if (ptr == &pool[i])
      {
        VALGRIND_MEMPOOL_FREE(pool, ptr);
           freeMap &= ~(1 << i);
        return;
      }
  }
}

int main (int argc, const char * argv[])
{
  MemPool mempool;
  int *ptrs[3];

  ptrs[0] = mempool.allocInt();
  ptrs[1] = mempool.allocInt();
  ptrs[2] = mempool.allocInt();

  mempool.freeInt(ptrs[0]);
  mempool.freeInt(ptrs[2]);
}

valgrind -v --leak-check=full--show-reachable=yes
./main

==9971== Searching for pointers to 1 not-freed
blocks
==9971== Checked 180,728 bytes
==9971==
==9971== 4 bytes in 1 blocks are still reachable
in loss record 1 of 1
==9971==    at 0x400C28: MemPool::allocInt()
(main.cpp:46)
==9971==    by 0x400D6C: main (main.cpp:73)
==9971==
==9971== LEAK SUMMARY:
==9971==    definitely lost: 0 bytes in 0 blocks
==9971==    indirectly lost: 0 bytes in 0 blocks
==9971==      possibly lost: 0 bytes in 0 blocks
==9971==    still reachable: 4 bytes in 1 blocks
==9971==         suppressed: 0 bytes in 0 blocks
```

Listing 9 (cont'd)

Black-Scholes in Hardware

The Black-Scholes model is a financial model. Wei Wang outlines its design and implementation for those who want to understand how algorithms can be implemented in hardware.

The Black-Scholes model is a mathematical model developed by F. Black and M. Scholes in the early1970s for valuing European call and put options on a non-dividend-paying stock [Hull06]. *European option* is a type of option that can be exercised only at the end of its life, whereas *American option* is another type of option that can be exercised at any time up to the expiration date. A *call option* gives the holder the right to *buy* an underlying asset by a certain date at a certain price. A *put option* gives the holder the right to *sell* an underlying asset by a certain date at a certain price. The date specified in the contract is known as the *expiration date* or the *maturity date*. The price specified in the contract is known as the *exercise price* or the *strike price*.

The Black-Scholes formula for the prices at time zero of a European *call* option on a non-dividend-paying stock is:

$$c = S_0 N(d_1) - Ke^{-rT} N(d_2) \tag{1.1}$$

and a European put option on a non-dividend-paying stock is:

$$p = Ke^{-rT} N(-d_2) - S_0 N(-d_1) = c + Ke^{-rT} - S_0 \tag{1.2}$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}} \tag{1.3}$$

$$d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \tag{1.4}$$

The variables c and p are the European call and put option price, S_0 is the stock price at time zero, K is the strike price, r is the continuously compounded risk-free interest rate, σ is the stock price volatility, and T is the time to maturity of the option, which is represented as: 3 months as 0.25, 6 months as 0.5, 1 year as 1.0.

The function $N(x)$ in (1.1) and (1.2) is the cumulative probability distribution function of a standard normal distribution. The probability function of a standard normal distribution is given by the following equation, which is the first-order derivative of the standard normal distribution density function $N(x)$.

$$P(x) = N'(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{1.5}$$

The only problem in implementing equations (1.1) and (1.2) is in computing the cumulative normal distribution function $N(x)$. This function can be approximated by a polynomial function that gives six-decimal-place accuracy:

$$N(x) = \begin{cases} 1 - N'(x)(a_1 k + a_2 k^2 + a_3 k^3 + a_4 k^4 + a_5 k^5), x \geq 0 \\ 1 - N(-x), x < 0 \end{cases} \tag{1.6}$$

where:

$$k = \frac{1}{1 + \gamma x}, \ \gamma = 0.2316419,$$

$$a_1 = 0.319381530, \ a_2 = -0.356563782, \ a_3 = 1.781477937,$$

$$a_4 = -1.821255978, \ a_5 = 1.330274429$$

The Black-Scholes model implemented in the PARSEC benchmark [Bienia11] is exactly as introduced in this section, and in the next section, the software implementation is from the PARSEC implementation with minor modifications [PARSEC], the benchmark also comes with synthetic test data inputs (portfolio) based on replication of 1,000 real options. The benchmark is coded in C/C++ with default single precision floating point. The benchmark implementation offers thread-level parallelism with Pthreads, OpenMP and Intel TBB, and runs on Linux, Solaris 10, and Windows platforms. The benchmark can be compiled with GCC 4.3 and ICC 10.1 to run on SPARC, i386, X86_64 and ARM CPU architectures.[1]

Software implementation of the Black-Scholes model

To compute a call or put option price in equations (1.1) and (1.2), we should first compute d1 and d2 in equations (1.3) and (1.4), and use the results to compute the standard normal distribution probability function in equation (1.5) and feed into the cumulative normal distribution function in equation (1.6), and then feed the results to compute the option price in equation (1.1) or (1.2). Following the flow of data, the model can be clearly divided into three sequential blocks: 1) D1D2, that is d1 from equation (1.3) and d2 from equation (1.4); 2) CNDF, the cumulative normal distribution function in equation (1.6); and 3) OP, the option price as in equation (1.1) and (1.2). The implementation of each function block with data inputs and outputs is shown below in sequence.

The D1D2 function takes five input parameters – spot price, strike price, interest rate, volatility and time-to-maturity – into computing equation (1.3) and (1.4), the results are returned into d1 and d2. (See Listing 1.)

The CNDF function implements cumulative normal distribution function in equation (1.5) and (1.6).The function takes d1 and d2 separately as its input and computes the cumulative normal distribution as its output (see Listing 2).

Wei Wang studied Engineering at Cambridge. Wei currently works in computer systems research, with interests in how software stacks run on CPUs and interact with memory systems and I/O. Over the past two+ years Wei has used C++ intensively for building a computer system simulator for performance evaluation. Wei can be contacted at w.wang.05@cantab.net

1. The PARSEC benchmark also includes another financial analysis application, the HJM (Heath-Jarrow-Morton) model to price swaptions, implemented in C++ with multithreading support for Pthreads and Intel TBB on Linux and Solaris 10 platforms. Due to the data-level parallelization of the workload, the performance scales well with the number of available cores on a CPU.

To run the multithreaded Black-Scholes application efficiently on a multicore CPU, the number of concurrent threads should match the number of cores

```
typedef float fptype;

void D1D2(

  //inputs
  fptype spotprice,
  fptype strike,
  fptype rate,
  fptype volatility,
  fptype time,

  //outputs
  fptype* d1,
  fptype* d2)

{
  fptype xSqrtTime = sqrt(time);
  fptype logValues = log(spotprice/strike);
  fptype xPowerTerm = volatility * volatility;
  xPowerTerm = xPowerTerm * 0.5;

  fptype xD1 = rate + xPowerTerm;
  xD1 = xD1 * time;
  xD1 = xD1 + logValues;

  fptype xDen = volatility * xSqrtTime;
  xD1 = xD1/xDen;
  fptype xD2 = xD1 - xDen;

  *d1 = xD1;
  *d2 = xD2;
}
```

Listing 1

The Black-Scholes equation takes seven input parameters, and computes the option price. The function implements equation (1.1) and (1.2) with calls to function D1D2 and CNDF (see Listing 3).

To run the multithreaded Black-Scholes application efficiently on a multicore CPU, the number of concurrent threads should match the number of cores to avoid unnecessary context switch, also use thread affinity to avoid unnecessary threads migration among different cores, and each thread should match its working sets size to the CPU cache and memory hierarchy. For example, one option input data entry can fit in one cache line of 64 bytes, a 64KB L1 cache can hold up to 1000 options, and while a 2MB L2 cache can hold up to a portfolio of 32 sets of 1000 options. As L1 access latency is a few (<10) cycles, L2 access latency is 10+ cycles, while L3 is usually shared among cores with 40 cycles access latency, and the off-chip memory takes more than 100 cycles to access, it makes sense to match the data sizes with the cache and memory hierarchy.

```
//Cumulative Normal Distribution Function
#define inv_sqrt_2xPI 0.39894228040143270286
fptype CNDF(fptype InputX)
{
  int sign;
  fptype OutputX;
  fptype xInput;
  fptype xNPrimeofX;
  fptype expValues;
  fptype xK2;
  fptype xK2_2, xK2_3;
  fptype xK2_4, xK2_5;
  fptype xLocal, xLocal_1;
  fptype xLocal_2, xLocal_3;
  //Check for negative value of InputX
  if (InputX<0.0){
    InputX=-InputX;
    sign=1;
  }else
    sign=0;
  xInput=InputX;

  // compute NPrimeX term common to both four &
  // six decimal accuracy calcs
  expValues = exp(-0.5f * InputX * InputX);
  xNPrimeofX = expValues;
  xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI;
  xK2 = 0.2316419 * xInput;
  xK2 = 1.0 + xK2;
  xK2 = 1.0/xK2;
  xK2_2 = xK2 * xK2;
  xK2_3 = xK2_2 * xK2;
  xK2_4 = xK2_3 * xK2;
  xK2_5 = xK2_4 * xK2;
  xLocal_1 = xK2 * 0.319381530;
  xLocal_2 = xK2_2 * (-0.356563782);
  xLocal_3 = xK2_3 * 1.781477937;
  xLocal_2 = xLocal_2 + xLocal_3;
  xLocal_3 = xK2_4 * (-1.821255978);
  xLocal_2 = xLocal_2 + xLocal_3;
  xLocal_3 = xK2_5 * 1.330274429;
  xLocal_2 = xLocal_2 + xLocal_3;
  xLocal_1 = xLocal_2 + xLocal_1;
  xLocal = xLocal_1 * xNPrimeofX;
  xLocal = 1.0 - xLocal;
  OutputX=xLocal;
  if(sign){
    OutputX = 1.0 - OutputX;
  }
  return OutputX;
}
```

Listing 2

the cumulative normal distribution function ... can be approximated by a polynomial function that gives six-decimal-place accuracy

	FPGA					GPU	Cell BE		CPU
	Floating point		Fixed point			Single	Double	Single	Double
	Double	Single	48bit	32bit	18bit				
4LUTs	55925	27793	36183	15757	8850	-	-	-	-
DSP blocks	124	31	83	48	12	-	-	-	-
Cores (LX/SX)	1 / 0	3 / 1	1 / 1	2 / 3	8 / 5	unknown	16	32	1
Clock (MHz)	67	61	49	64	81	400	3200	3200	2500
Least[2] error	$2x10^{-5}$	$4x10^{-3}$	$8x10^{-5}$	$8x10^{-5}$	$6x10^{-3}$	$4x10^{-3*}$	$8x10^{-5*}$	$4x10^{-3}$	$8x10^{-5}$
Acceleration (x)	15	41	11	29	146	32*	5	29	1

Table 1

FPGA based accelerators for financial applications

There are a few companies offering FPGA-based accelerators for computing the Black-Scholes model and Monte-Carlo simulation for pricing options, such as Celoxica [Morris07] and Maxeler [Richards11].

Celoxica had implemented FPGA based acceleration technologies for European options pricing. They achieved 15 times speed-up over an existing server at full precision and have similar performance to GPU and Cell implementations as shown in the table below [Morris07]. The accelerations achieved by FPGA, GPU and Cell BE are compared against the fully optimized C++ implementation running on a PC with a single core AMD 2.5GHz Opteron processor with 2 Gb of RAM and the Windows 2000 OS.

Table 1 shows a comparison of resource utilization, error and acceleration for different implementations of European option benchmark. In the table, LX/SX stands for two FPGA devices from the Xilinx Virtex 4 family, the LX160 and the SX55. The FPGA clock rates and accelerations are given for the LX device. Results indicated by * are estimates. The SX variant of the Virtex 4 family is significantly richer in DSP blocks resources, at the expense of fewer 4LUTS. The speed grade chosen for both devices was at the same -10 speed grade.

The component implemented in the FPGA is the computation unit for computing the following payoff equation (1.7). The computationally intensive component of computing the payoff equation is the Gaussian Random Number Generator, as Z_n is generated by the Gaussian Random Number Generator (GRNG). The other components other than the GRNG for computing the above equation are just multipliers, adder, natural exponent, subtractor, max and accumulator.

$$\sum_{i=1}^{n} \max(0, S(0)e^{\bar{x}+\sigma Z_n} - K) \tag{1.7}$$

The payoff equation is implemented in HyperStreams that is built on the Handel-C[2] programming language. The data flow and the control flow of the implementation are separated, the data flow is programmed using the HyperStreams abstraction, and the control flow is programmed using traditional Handel-C syntax. The designs were synthesized using Celoxica DK5 and Xilinx ISE 9.1.

The block diagram in Figure 1 shows the portion of the European option-pricing algorithm implemented on FPGA, noting the separation of control and pipelined data flow. The parameters provided from the control flow to the data flow are fixed constants during the computation of the above equation and are therefore calculated in software.

As the FPGA designs are implemented in the high-level abstraction Handel-C programming language rather than implemented in RTL, it's not a difficult task to implement the design in different flavours of floating point and fixed point. Balancing the resource utilization, performance and precision, the 32-bit fixed-point implementation offers the best results. The 18-bit fixed-point implementation offers 146 times performance acceleration but has 133 times worse precision compared to CPU as shown in Table 1, the single floating-point implementation offers 41 times acceleration but has 200 times downgrade on precision.

The power consumption and cost have not been taken into account when comparing the performances of different implementations. The Handel-C approach has a clear advantage on the time-to-market metric, as the five different flavours of floating point and fixed-point implementations only took two person days to implement. However, this approach doesn't work out-of-the-box with legacy C/C++ code base, which limits its potential.

Maxeler worked with J.P. Morgan Quantitative Research to accelerate their tranche valuation [Richards11]. The base correlation with stochastic recovery model is used to price and calculate risk for tranche-based products, such as vanilla tranches, bespoke tranches, n-th to default and

2. Handel-C is a programming language and is not a Hardware Description Language (HDL) for compiling programs into hardware images of FPGAs or ASICs. It is a rich subset of C, with non-standard extensions to control hardware instantiation and parallelism.

Following the flow of data, the model can be clearly divided into three sequential blocks

CDO^2. At its core, the model involves two key computationally intensive loops of constructing 1) the conditional survival probabilities using a Copula as shown in Equation (1.8) and 2) the probability of loss distribution using convolution as shown in Figure 2. Inside the convolution, FFT is used to evaluate the integral:

$$g_\rho\left(p_i,u\right)=N\left(\frac{N^{-1}\left(p_i\right)-\sqrt{\rho}M}{\sqrt{1-\rho}}\right) \tag{1.8}$$

where g_p is the conditional survival probability for this name, p_i is the unconditional survival probability for this name, ρ is the correlation and M is the market factor.

The valuation of tranched CDOs can be expressed in flattened C code as below after removing all use of classes, templates and other C++ features in order to simplify parallelization. The Copula takes 23% of execution time and the Convolution takes 75% of execution time in CPU. (Listing 4.)

After offloading the computation of Copula and Convolution onto the FPGA from the CPU, a single FPGA prices a complex trade 134 times

```
//OptionPrice
fptype BlackScholes(fptype spotprice,
   fptype strike, fptype rate, fptype volatility,
   fptype time, int otype, float timet)
{
  fptype OptionPrice;
  fptype FutureValueX;
  fptype NofXd1;
  fptype NofXd2;
  fptype NegNofXd1;
  fptype NegNofXd2;
  fptype d1;
  fptype d2;

  //D1D2
  D1D2(spotprice, strike, rate, volatility,
       time, &d1, &d2);

  //CNDF
  NofXd1 = CNDF(d1);
  NofXd2 = CNDF(d2);

  //OP
  FutureValueX = strike * (exp(-(rate)*(time)));
  if (otype==0) {
    OptionPrice = (spotprice * NofXd1) -
                  (FutureValueX * NofXd2);
  }else{
    NegNofXd1 = (1.0 - NofXd1);
    NegNofXd2 = (1.0 - NofXd2);
    OptionPrice = (FutureValueX * NegNofXd2) -
                  (spotprice * NegNofXd1);
  }
  return OptionPrice;
}
```

Listing 3

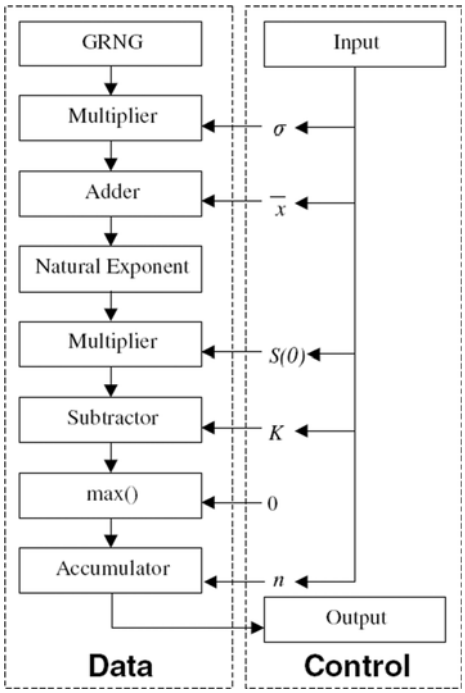


Figure 1

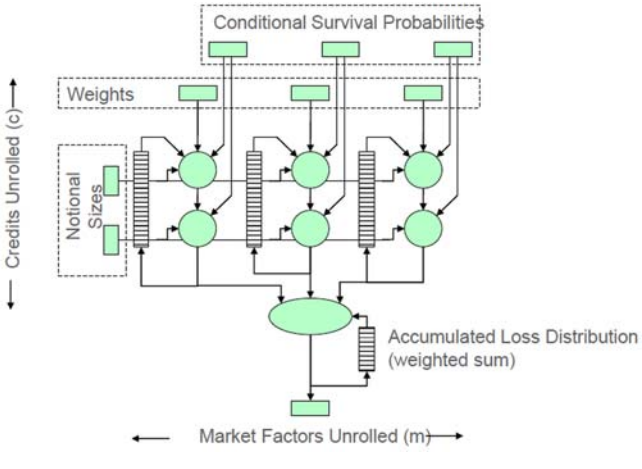


Figure 2

Derivatives pricing is at the core of financial trading and risk management

faster than a single CPU. As a result, end-to-end time to price global credit hybrids portfolio once reduced to ~125 seconds with pure FPGA time of ~2 seconds to price ~30,000 tranches and total compute time of ~30 seconds. End-to-end time for pointwise credit deltas on global credit hybrids portfolio reduced to ~238 seconds with pure FPGA time of ~12 seconds, using a 40-node FPGA machine. End-to-end time to run multiple trading/risk scenarios for desk reduced to ~320 seconds with results accurate to within $5 across global portfolio, while it's not previously possible to run such scenarios multiple times within a single trading day.

In addition to acceleration, the FPGA based solutions have predictable performance for computation and data I/O, as FPGAs are statically scheduled and with no cache involved. However, JPMorgan took a 20% stake in Maxeler, which potentially limits its adoption in other financial institutions.

FPGA based high performance computing for financial applications

Tandon [Tandon03] completed a Master's Thesis on *A Programmable Architecture for Real-Time Derivative Trading*, which he implements the Black-Scholes European Option Pricing model on FPGA, simulated ARM processor and Mathematica, which is used as the reference platform, and compares their performance acceleration and accuracy. The results are shown in Table 2.

In Table 2, time per iteration means the time used to compute either a call or put option price using the Black-Scholes model given a set of input data. The Reference Mathematica test is conducted on Mathematica 5.0 on an Intel Pentium 4 processor at 2.53 GHz. The Black-Scholes model is implemented in Mathematica using some of its library functions that are assumed to have suitable optimizations or approximations. Floating point is used in this implementation, but the thesis doesn't tell whether it being single or double floating point. The simulated ARM processor is done on a simulated ARM7TDMI processor running at 200MHz, the Black-Scholes model is implemented in ANSI-C with floating point and targets towards an ARM7 processor. The simulated ARM7TDMI simulates all floating-point computations within the processor itself rather than having a dedicated floating-point unit. The FPGA based implementation coded in VHDL[3] has not been synthesized successfully due to it being a purely floating-point computation and the IEEE math library that is used for the floating-point computations is designed for simulation only. However, modifications have been made to the design to make it integer based, the performance numbers are drawn from the integer-based implementation of the Black-Scholes model on a Virtex-II Pro FPGA. The 50ns time per iteration number shown in the table, however, is not measured from real experimental hardware rather it is an estimated number inferred from the synthesis report of the design. The downgrade from floating point to integer-based implementation significantly undermines the accuracy.

```
for i in 0 ... markets-1
  for j in 0 ... names-1
    prob = cum_norm((inv_norm(Q[j])
                     -sqrt(p)M)/sqrt(1-p);
    loss = calc_loss(prob,Q2[j],
                     RR[j],RM[j])*notional[j];
    n = integer(loss);
    L = fractional(loss);
    for k in 0 ... bins-1
      if j == 0
        dist[k] = k == 0 ? 1.0 : 0.0;

      dist[k] = dist[k]*(1-prob) +
                dist[k-n]*prob*(1-L) +
                dist[k-n-1]*prob*L;
      ifj==credits -1
        final_dist[k] += weight[i] * dist[k];
    end # for k
  end # for j
end # for i
```

Listing 4

The challenges faced in the Black-Scholes model FPGA implementation using floating point in the thesis however points out that a fixed-point implementation of the Black-Scholes model on FPGA is more favourable considering the manpower required to implemented the floating-point capability and the accuracy tradeoff between floating point and fixed point.

It is also pointed out that financial models are very heavily dependent on calculus, probability, statistics and other branches of Mathematics, a logic library which has RTL implementations of some fundamental mathematical functions would be very useful, such as, integration, higher order derivation, random number generation, statistical and stochastic modelling, vector calculus, trigonometric functions and logarithmic functions. Although the idea is constructive for putting more financial models on FPGAs easily, it should be noted that integration is not necessary for calculating cumulative normal distribution function in the Black-Scholes model, as a polynomial approximation that gives six-decimal-place accuracy is given in [Hull06].

Black-Scholes hardware design

The Black-Scholes model can be similarly implemented in three hardware modules: D1D2, CNDF and OptionPrice, as shown in Figure 3. D1D2 module computes Equations (1.3) and (1.4); it takes five data inputs and

Experiment platform	Time per iteration	Accuracy
Mathematica -Reference	15.625 µs	Very high
Simulated ARM processor	170 µs	High
Reconfigurable logic	50 ns	Medium high

Table 2

3. VHSIC Hardware Description Language

The downgrade from floating point to integer-based implementation significantly undermines the accuracy

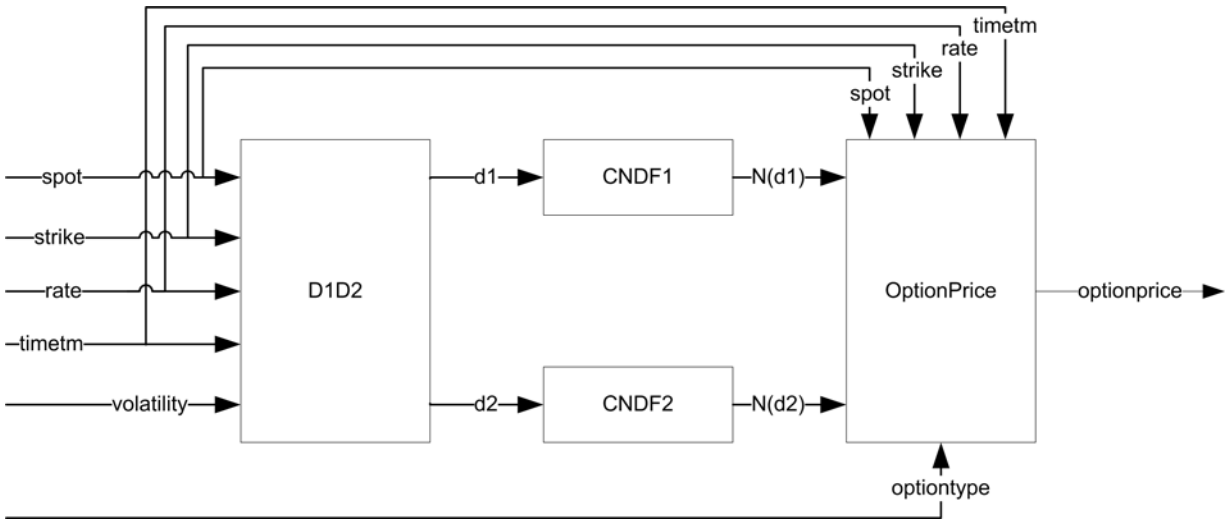


Figure 3

then feed the two outputs to the two parallel CNDF modules. CNDF (Cumulative Normal Distribution function) module computes Equation (1.5) and (1.6); it takes the input from D1D2 and feeds the output to OptionPrice module. OptionPrice computes Equation (1.1) and (1.2); it takes four data inputs, one option type control signal and two data feeds from CNDF modules, the module gives the price of the option as the output. The implementation takes 25 clock cycles to compute the option price based on the five data inputs and one option type control input, with each arithmetic unit taking only one cycle to compute for simplicity[4]. The data path resource utilization and the time delay of each module are summarized in Table 3.

The resource utilization count assumes each hardware arithmetic unit is shared among D1D2, CNDF and OptionPrice blocks where possible. In a fully pipelined implementation, we would see at the bottom row the sum of each column rather than the maximum of each column, for example, in column +, it would be three + hardware arithmetic units rather than one unit to be needed for the implementation. The clock cycle count can be seen visually in the block diagrams as shown in Figure 4, Figure 5, Figure 6, each horizontal level represents one cycle delay.

	Arithmetic Unit							
	+	−	×	÷	√	e^x	ln(x)	Cycles
D1D2	1	1	2	1	1	0	1	7
CNDF	1	1	3	1	0	1	0	12
OptionPrice	1	1	2	0	0	1	0	6
BlackScholes	1	1	3	1	1	1	1	25

Table 3

In the following sections, the implementation details of D1D2, CNDF and OptionPrice blocks are explained.

D1D2 block design

This block takes 7 cycles to execute, it has 1 add unit, 1 subtract unit, 2 multiply units, 1 divide unit, 1 square root unit and 1 logarithm unit, as shown in Figure 4. The inputs to the block are spot price, strike price, time to mature, volatility and interest rate, which are shown on the top of Figure 4, and the outputs of the block are d1 and d2, which are shown at the bottom of Figure 4. The input from the right of the block diagram is the control signal to the data path; it is a constant in this case.

CNDF block design

This block takes 12 cycles to execute, it has 1 add unit, 1 subtract unit, 3 multiply units, 1 divide unit, 1 exponential unit, as shown in Figure 5. The inputs on the top of the block diagram are inputs to the module and the outputs at the bottom of the block diagram are the outputs of the module. The inputs from the right of the diagram are control signals to the data path.

OptionPrice block design

This block takes 6 cycles to execute, it has 1 add unit, 1 subtract unit, 2 multiply units and 1exponential unit as shown in Figure 6. The inputs on the top of the diagram are inputs to the module and the outputs at the bottom

4. The one cycle implementation is not area efficient and cost effective, the more complex arithmetic units, such as divide, square root, and logarithm and exponential, take more than 20 cycles to compute in practice.

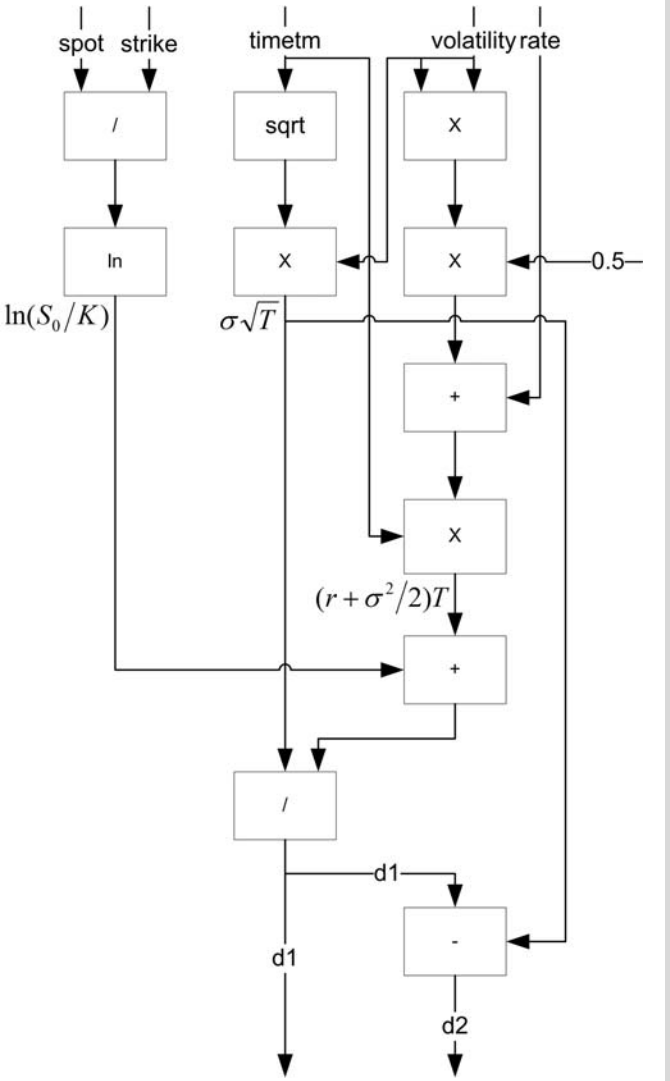


Figure 4

of the diagram are the outputs of the module. The inputs from the right of the diagram are control signals to the data path.

Black-Scholes hardware implementation

The hardware implementation is based on single precision floating point, as the baseline implementation in PARSEC is in single precision floating point. The decision to implement the model whether in single-precision floating point or 32-bit/18-bit fixed point depends on the efforts to implement, the logic resource requirement, and the speed of acceleration and the accuracy. The floating-point arithmetic units make use of the components from the Synopsys DesignWare floating-point datapath library [Synopsys] and the Synopsys Synplify Premier 2010.09 for FPGA implementation. DesignWare floating-point library contains all the components needed to implement the floating point arithmetic functions of +, -, *, /, sqrt, exp and ln in the Black-Scholes model. The Synopsys DesignWare library is optimized for ASIC implementation rather than FPGA, though the design efforts are minimal, the area of the design is huge when synthesized to FPGA.

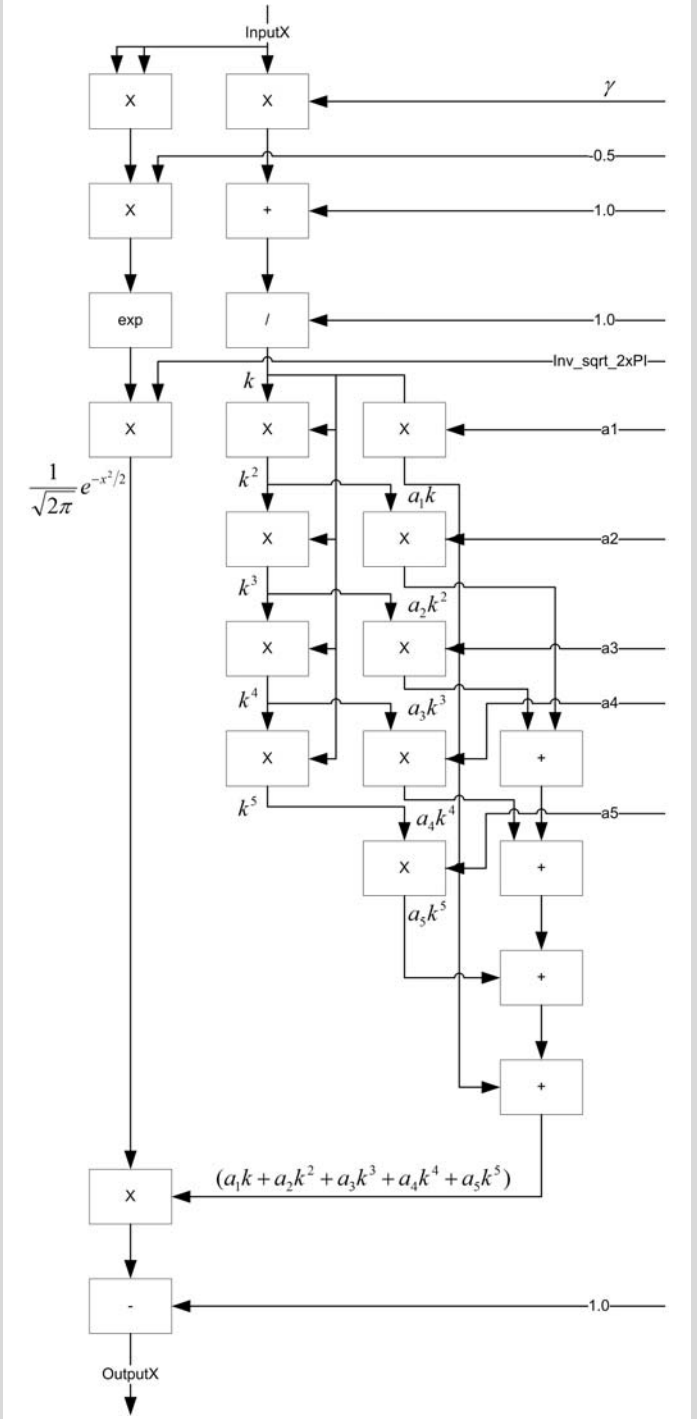


Figure 5

An area optimized floating point library of square root, logarithm and exponential components was later developed specifically for FPGA, which showed significant improvements in implementation area and fits the whole design onto a tiny Xilinx Spartan 3A FPGA as shown in Table 4. The Add/Sub/Mult/Div uses the Xilinx floating-point operators. The square root and logarithm are implemented CORDIC rolled [Wikipedia11], while the exponential is implemented using table lookup.

	4-input LUTs	MULT	BRAM (Kbit)	Clock freq (MHz)	Clock cycle	Throughput (KFLOPS)
Original	34747 (295%)	20	90.11 (5 blocks)	8.2	25	328
New	10228 (86%)	20	92.16 (5 blocks)	58.54	277	211

Table 4

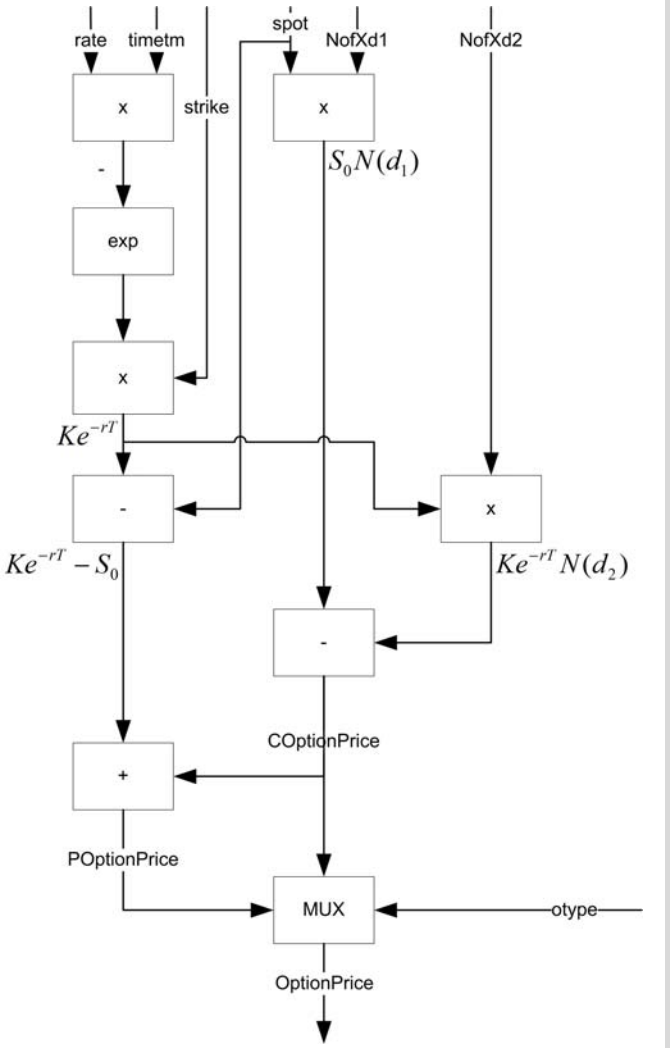


Figure 6

Conclusion

Derivatives pricing is at the core of financial trading and risk management. As shown in [Richards11], FPGA offers the opportunity for real-time risk visibility to monitoring and controlling financial risk of complex derivatives that are not possible on CPUs.

The intention for this article is to show software engineers how an algorithm can be implemented in software and hardware. As each arithmetic unit matches to a hardware component, so the software implementation is intentionally coded like how it should be implemented in hardware. The idea is that it should be very easy to match the hardware block diagrams to the corresponding software function implementations.

The Black-Scholes model serves as a baseline of all financial models for pricing derivatives, most of these financial models rely on the same floating point computations of the seven basic arithmetic operations: add, subtract, multiply, divide, square root, logarithm and exponential. The intention is that it should be very easy to move from Black-Scholes to another financial model, such as HJM for pricing swaptions or base correlation with stochastic recovery for pricing tranches. ■

References

[Bienia11] Christian Bienia, *Benchmarking Modern Multiprocessors*, Princeton University, New Jersey, PhD Thesis 2011.

[Hull06] John C. Hull, *Options, futures and other derivatives*, 6th ed. New Jersey, U.S.A: Prentice-Hall, 2006, pp. 295-298.

[Morris07] Gareth W. Morris and Matt Aubury, 'Design Space Exploration of the European Option Benchmark Using HyperStreams' in *Field Programmable Logic and Applications* (FPL), 2007., Amsterdam, 2007, pp. 5-10.

[PARSEC] Source code available at: http://parsec.cs.princeton.edu/

[Richards11] Peter Richards and Stephen Weston. (2011, May) Stanford EE Computer Systems Colloquium.[Online]. http://www.stanford.edu/class/ee380/Abstracts/110511.html

[Synopsys] Synopsys Inc. 'Datapath – Floating Point Overview.'

[Tandon03] Sachin Tandon, *A Programmable Architecture for Real-Time Derivative Trading*, Computer Science, University of Edinburgh, Edinburgh, MSc Thesis 2003.

[Wikipedia11] Wikipedia. (2011, Feb) http://en.wikipedia.org/wiki/CORDIC.

Replace User, Strike Any Key?

There is a common perception in the IT industry that the user is the primary source of all the problems. Sergey Ignatchenko asks if this is true.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with opinions of the translator or Overload editors; please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry2004]) might have prevented from providing an exact translation. In addition, both translator and Overload expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Our jobs would be so much easier
if it weren't for all those pesky users.
~ [ComputerWeekly2010]

There is a common perception in IT industry that user is the primary source of all the problems. In a sense, it is true: if not for users, there wouldn't be any IT jobs, and therefore won't be any IT problems; I am not sure though if this is what IT professionals should really want. While having too many problems on the job is not that good, having one single problem of 'how to find the job' is IMNSHO a significantly worse alternative.

The question of relations between users and developers has already been touched in [NoBugs2011], which establishes that it is the user who has the upper hand in the user-developer relationship, and that it is responsibility of developer to make the user happy. This article aims to analyze the problem in more detail and from different angles.

There is always somebody else to blame

Nihil humani a me alienum puto
(Nothing human can be alien to me)
~ Publius Terentius Afer, 2nd century BC

First of all, let's start with a trivial observation: rabbits (as well as people) in general are rarely willing to admit their own mistakes; one very good book on this subject is [MistakesWereMade]. Here we will not go into details of this phenomenon, but will merely admit that IT professionals (including us, developers), are still human (or sometimes rabbit) beings, and therefore we have a natural tendency to blame the others for our own mistakes. In the case when we didn't expect that the user will press that specific button in that specific situation, it is very natural to shout 'You need to be an idiot to <place whatever we didn't expect user to do here>!!!'

In fact, the tradition of blaming users for whatever happens to what we've made, is not specific to IT. For example, famous Murphy's Law has originated when Edward Murphy blamed a technician from MX981 team for incorrectly connecting sensors for the system Murphy designed [HistoryOfMurphysLaw]; it obviously didn't cross Murphy's mind (at least not at that time) that a robust system should have connectors which don't allow for incorrect connection. This tradition of blaming users for our own mistakes has flourished in IT world.

On reasonable expectations

If I ordered a general to fly from one flower to another like a butterfly, ... and if the general did not carry out the order that he had received, which one of us would be in the wrong? ... The general, or myself?
~ Antoine de Saint-Exupéry, The Little Prince

One huge mistake developers (and even business analysts) tend to make, is that they expect users to behave rationally at all times. Nothing could possibly be further from reality. From time to time, everybody is entitled to make a mistake; and users are entitled to make them much more often than developers, because usually it is users who're paying.

Even if developer is right 99% of the time, he's still wrong in 1% of cases. This means that even if users make the same percentage of mistakes (which is, as explained above, a very optimistic estimate), and if your application has 1 million users making 100 operations per day each, you'll still get 1 million mistakes per day made by users. Among those mistakes, most will be trivial, but given the sheer volume of attempts, users will almost certainly try pretty much every erroneous scenario, including those we have not thought about. Who is here to blame? IT tradition assumes that 'it is those stupid users'; users (sometimes joined by management) tend to say 'it is those idiot developers'. In this debate, we take the third position: we say that the blame is on those rabbits who expected that under the circumstances, mistakes (on both sides) won't happen.

Whenever a system of this scale is first launched, it is reasonable to expect that some of erroneous scenarios won't be covered in original release, and to be prepared to identify problems and to fix them within reasonable (for the users, not for the developers) time frame. Still, while expecting flawless programs from the very beginning is not exactly reasonable, it is important to realize that while some bugs are inevitable, they still are bugs (and not users fault), and therefore must be fixed.

In addition, it should be noted that in many cases, mistakes are direct result of the fact that developers and users have very different perspectives of the software, which results in miscommunications. We feel that it is the developer's responsibility at least to try to look at the program from user's point of view, which leads us to...

Trying on the user's hat

He that increaseth knowledge increaseth sorrow.
~ Ecclesiastes, 1. 18

Another important thing for the developer is understanding how users will use the program. And here developers tend to have major problems. In [NoBugs2011] it has been noted that developers are notoriously bad in creating UIs; here we will elaborate on it.

In fact, it seems that developers are not only bad in creating UIs, but are also bad in any task which needs the programmer to put on the user's shoes.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

One should never ever try to formalize things at a level which one doesn't understand

So, we asked ourselves: is it due to developers being inherently different or because of them already being involved with the project in another role?

To find it out, we've tried a small-scale experiment with a few of our fellow rabbits. We asked the very same developers who were notorious for creating pretty bad UIs, to design UI for a project where developers were not involved. The result was rather obvious (though due to small scale it is unclear if it is statistically significant, and further research is suggested): the very same rabbits who designed bad UIs when they were involved as developers, created very decent UIs when they were designing UIs while being completely in user's shoes, in particular, not being involved in the project in any other way, and without any knowledge about implementation.

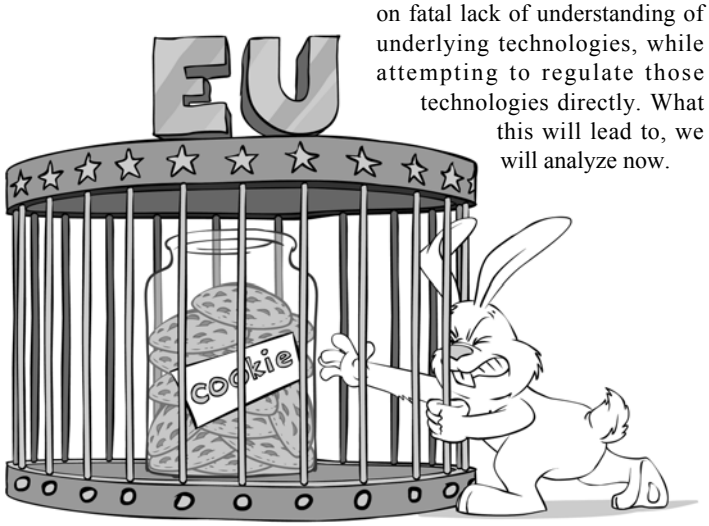
If further research will confirm this hypothesis, it will mean that it is knowledge about system implementation which causes developers to fail to look at things from user's perspective. It calls for creating a separate team of business analysts (BAs); while this practice is already rather common in the industry, what is new is that our research suggests that the same rabbit might be able to work both as a developer and as a BA, as long as her work as a developer and as BA occurs only in completely separate projects.

EU cookie directive

The road to hell is paved with good intentions
~ proverb

For a long time we thought that developers were the worst rabbits to design UIs. Recently, we've found there are rabbits out there who can do even worse, and they are bureaucrats. One recent example is an infamous EU Cookie Directive (strict name is 'Directive on Privacy and Electronic Communications', but here we will be dealing with one aspect of it, namely with websites being required to ask consent of users before storing a cookie on users' computers ([2009/136/EC], [ICOGuidance])).

This whole document (as well as preceding directive 2002/22/EC) is based on fatal lack of understanding of underlying technologies, while attempting to regulate those technologies directly. What this will lead to, we will analyze now.



Without going into details of legaleze in the related documents, what it essentially required from web sites (in UK – starting from May 2012), is to ask user confirmation before 'storing' a cookie on an end-user's computer. While obviously well-intended (the idea was to protect users' privacy), both proposed requirements and their interpretations are fatally flawed. Essentially, what is required is to ask a user before site placing any cookie; many sites have already started changing UIs just to comply with the directive. What will happen when enough sites implement it, is obvious: when users will get used to such requests (usually phrased as 'to access this site, you need to enable cookies, please confirm <yes>/<no>'), users will start pressing 'Yes, I want a cookie' button every time they see it. This phenomenon (known as capture errors) is well-known in security industry (see, for example, [SecurityEngineering], section 2.3.1), and is clearly unavoidable here. As soon as it happens, the whole point of directive would be lost, and it will merely create a nuisance for users, without any perceivable benefit.

This is not the only problem with the directive. Whoever made it, has tried to think about it a bit further; unfortunately, without understanding technology involved, an attempt to formalize requirements at technology level has had an exactly opposite effect. The directive provides for an exemption for 'where such storage or access is strictly necessary for the provision of an information society service requested by the subscriber or user'. Once again, intentions were good. Unfortunately, 'strictly necessary' wording makes it perfectly useless (and actually, even worse than that, as described below). As we know, strictly speaking, cookies are never 'strictly necessary' (this is because you can always, for example, put all information you need, into dynamic URL; it is a major hassle, but it is still possible, therefore alternatives are not 'strictly necessary'). Now things begin to become even worse. The Information Commissioner's Office has provided an interpretation of the EU directive, where 'strictly necessary' is not really 'strictly' necessary, but 'essential, rather than reasonably necessary' [ICOGuidance]; here 'strictly necessary' has degraded to 'essential' (which is still not exactly defined). What this travesty means in practice, is that those who will interpret it on a cautious side, will still ask for a confirmation, and will annoy their users, losing business and money; and those who don't care about privacy at all, will improve their business even further. The whole result seems to be an exact opposite of good intentions behind the directive.

One should never ever try to formalize things at a level which one doesn't understand. To do a reasonably good job, members of the European parliament have had two options: a) to specify privacy requirements without going into this level of details, and avoiding reference to specific technologies (admittedly, they've tried to, but level of requirements they've chosen, was apparently still too low), or b) to understand how cookies really work, and to take several less drastic and more reasonable measures, including, probably, a prohibition on third-party cookies (which is where most privacy leaks reside). In fact, members of European parliament have decided to take a middle ground between these two options, which (as we've seen above) has failed miserably.

any decision should belong to the one who is in better position to make it, and very often it is the developer who has a better understanding of the issue in hand

Options: why more is less

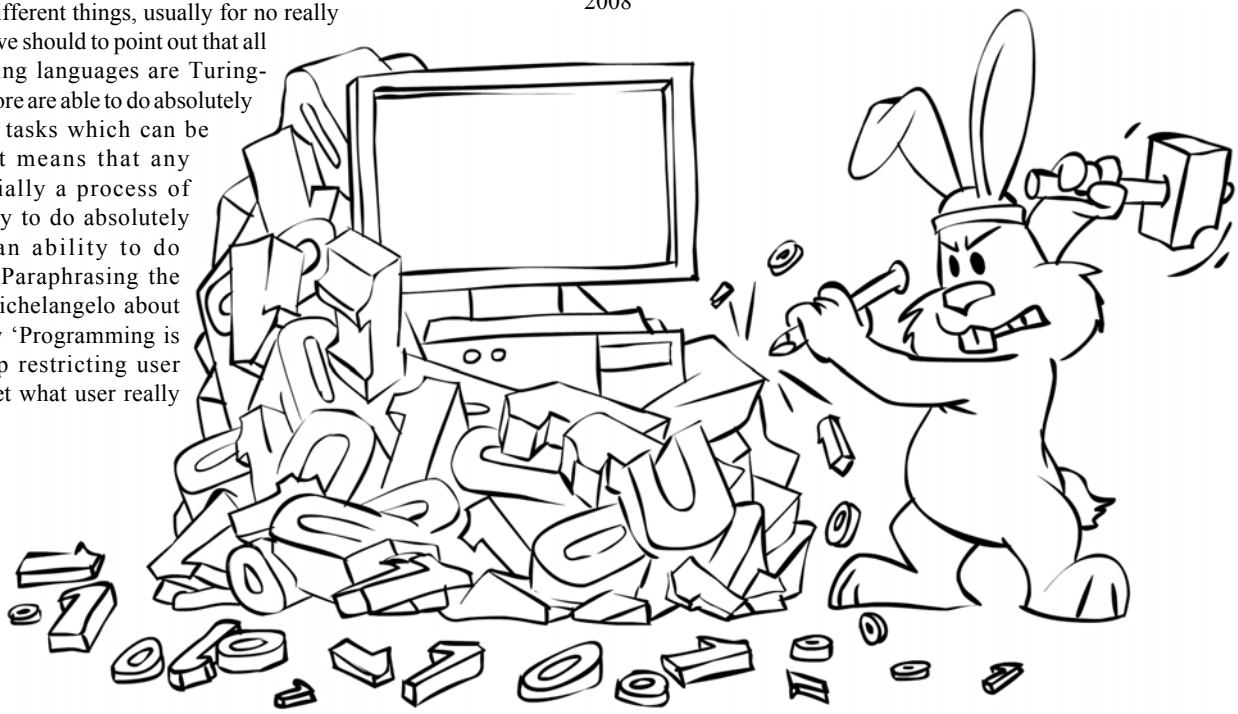
Carving is easy, you just go down to the skin and stop.
~ Michelangelo

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.
~ Antoine de Saint-Exupéry

Actually, there is one thing useful about EU Cookie Directive: it shows us that shifting responsibility to the user is not always a good thing. In fact, it is rarely a good thing. Obviously, for a developer it is always very convenient, if he has any doubts, just not to make any decisions, and provide user with an option, to shift responsibility from developer to the user.

In practice, such 'passing the buck' is often not a good idea for one simple reason: because any decision should belong to the one who is in better position to make it, and very often it is the developer who has a better understanding of the issue in hand. This is especially true when we're speaking about technical side of the program: asking user questions 'what do you want to pay for' is clearly a user question, but asking 'how much cache do you want to use for this program' is a developer question, whether we like it or not. In addition, shifting responsibility to user contributes to anxiety of customers related to having too many choices (for details, see [ParadoxOfChoice] book).

Still, developers often 'pass the buck' merely because they don't want any responsibility (with a common argument being 'I've already provided you with all the options, what else do you want?'). Unfortunately, this tendency is often aggravated by pressure from users (usually via managers) who are asking for slightly different things, usually for no really good reason), Here we should to point out that all modern programming languages are Turing-complete, and therefore are able to do absolutely everything (out of tasks which can be possibly done). It means that any program is essentially a process of reducing this ability to do absolutely everything into an ability to do something useful. Paraphrasing the famous quote of Michelangelo about carving, we can say 'Programming is easy, you just keep restricting user choices until you get what user really needs and stop'. ■



References

[2009/136/EC] DIRECTIVE 2009/136/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 25 November 2009 amending Directive 2002/22/EC on universal service and users' rights relating to electronic communications networks and services, Directive 2002/58/EC concerning the processing of personal data and the protection of privacy in the electronic communications sector and Regulation (EC) No 2006/2004 on cooperation between national authorities responsible for the enforcement of consumer protection laws

[ComputerWeekly2010] Users remain the weakest link in the IT security chain http://www.computerweekly.com/blogs/editors-blog/2010/03/users-remain-the-weakest-link.html

[HistoryOfMurphysLaw] *A History of Murphy's Law* Nick T. Spark, 2006

[ICOGuidance] Guidance on the rules on use of cookies and similar technologies, Information Commissioner's Office

[Loganberry2004] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[MistakesWereMade] *Mistakes Were Made (But Not by Me): Why We Justify Foolish Beliefs, Bad Decisions, and Hurtful Acts* Carol Tavris, Elliot Aronson, 2007

[NoBugs2011] The Guy We're All Working For , *Overload* #103

[ParadoxOfChoice] *The Paradox of Choice: Why More Is Less* Barry Schwarz, 2004

[SecurityEngineering] *Security Engineering* 2nd Edition, Ross Anderson, 2008

Simple Mock Objects for C++11

New C++11 features can be used to implement mock objects for unit tests. Michael Rüegg shows us how he does this in Mockator.

In our last article 'Refactoring Towards Seams in C++' (appeared in issue 108), we described how we can break dependencies in legacy code by applying seams with the help of our engineered refactorings. Once we have managed to apply seams, our code is not relying on fixed dependencies anymore, but instead asks for collaborators through dependency injection. Not only has our design greatly improved, but we are now also able to write unit tests for our code.

Sometimes it is impractical or impossible to exercise our code with real objects. If a real object supplies non-deterministic results, is slow or contains states that are difficult to create, then we might want to use mock objects to test our objects in isolation. In this article we present how we can mock objects by creating a small but useful mock object library that makes use of the new language features of C++11.

How to mock objects

To start with an example, consider the system under test (SUT) **`Trader`** shown in Listing 1. **`Trader`** has a fixed relationship to **`Nasdaq`** which makes it hard to test in isolation because its operations require network calls which we do want to avoid when we run our unit tests. **`Nasdaq`** is therefore a good candidate to be replaced with a mock object.

One of the many possibilities in C++ to inject dependencies from outside is to extract a template parameter and to inject the dependency at compile time making use of parametric polymorphism. We therefore call this seam type *compile seam*. After applying the refactoring *extract template parameter*, the code results as shown in Listing 2.

This code has a seam because we now have an enabling point: the place where the template class **`TraderT`** is instantiated. Note that we create a **`typedef`** which instantiates the template with the concrete type that has been used before applying the refactoring (here through the use of a default template parameter). This has the advantage that we do not break existing code that still wants to use **`Nasdaq`**.

```
#include "nasdaq.h"
struct Trader {
  void stopLoss(std::string symbol,
                unsigned int amount,
                Price lowerLimit) {
    Share share = exchange.lookupBy(symbol);
    // causes network call
    Price currPrice = share.currentPrice();
    if (currPrice < lowerLimit)
       exchange.sell(share, amount);
    // dito
    }
  }
private:
  Nasdaq exchange;
};
```

Listing 1

What is a seam?

A seam is a place in the code where we can alter behaviour without being forced to edit it in that place [Feathers04]. Every seam has one important property: an enabling point. This is the place where we can choose between one behaviour or another. There are different kinds of seam types. C++ supports object, compile, preprocessor and link seams.

We now give a complete example of how we think mocking objects should be done in C++ and explain the internals of our approach in the subsequent sections of this article. We apply the classic unit test work flow proposed by the xUnit pattern [Meszaros07] in Listing 3: setup, exercise, verify and teardown (whereas the latter is not necessary here).

We use the vector **`allCalls`** to register all function calls the SUT makes on the injected local class **`MockExchange`**. It needs to be defined static because of the shortcomings local classes still have with C++11. We create the vector initially with a size of one. Index 0 is reserved for calls of static member functions on the mock object. Note that every mock object has a mock ID which is used to access the calls made by the SUT on a specific instance of the mock object class. In the registrations of the function calls, we use this to store the call for the corresponding mock object instance. Also note the usage of C++11's new initialiser lists for specifying our expectations. At the end of our example, we assert the calls made with the index 1 (we only have one instance of **`MockExchange`**) with our expectations.

We think it is worthwhile to have the code for the mock object in the unit test without hiding it behind DSL's built up of macros as other mock object libraries do. This yields more transparency and exploits the full power of the host language when the library does not provide a desired feature. Furthermore, we circumvent the numerous problems that come with the application of macros.

```
#include "nasdaq.h"
template<typename STOCKEXCHANGE=Nasdaq>
struct TraderT {
  void stopLoss(std::string symbol,
     unsigned int amount, Price lowerLimit) {
    // as before
  }
private:
  STOCKEXCHANGE exchange;
};
typedef TraderT<> Trader;
```

Listing 2

Michael Rüegg is a scientific assistant at the Institute for Software of University of Applied Sciences Rapperswil. Mockator was the result of his master's thesis under the supervision of Prof. Peter Sommerlad.

An important part of a mock object implementation is the recognition of the function calls the SUT makes on the mock object while a unit test runs

```
#include "mockobjects.h"
#include <cassert>
void
test_sell_shares_when_current_price_below_stop_los
s_limit() {
  // setup
  static std::vector<calls> allCalls{1};
  struct MockExchange {
    MockExchange() :
       mockid{reserveNextCallId(allCalls)} {
      allCalls[mockid].pushback
         (call{"MockExchange()"});
    }
    Share lookupBy(std::string symbol) {
      allCalls[mockid].pushback
         (call{"lookupBy(std::string)", symbol});
      return {Share{symbol, Price{29}};
    }
    void sell(Share share, unsigned int amount) {
      allCalls[mockid].pushback
         (call{"sell(Share, unsigned int)",
                    share, amount});
    }
    const size_t mockid;
  };
  // exercise
  TraderT<MockExchange> trader;
  trader.stopLoss("FB", 1000000, Price{30});
  // verify
  calls expected = {
    {"MockExchange()"},
    {"lookupBy(std::string)", "FB"},
    {"sell(Share, unsigned int)", "FB", 1000000}
  };
  assert(expected == allCalls[1]);
}
```

Listing 3

In the classic mock object approach the unit test does not exercise any assertions. This is entirely handled by the mock object which – when called during SUT execution – compares the actual arguments received with the expected arguments using equality assertions and fails the test if they do not match. We have decided against this common approach and exercise the assertions in the unit test itself because we want to be independent of the underlying unit testing framework. We therefore do not assert for equality in the mock object member functions, but instead compare the string traces in the unit test. Also note that our comparisons are order-sensitive and therefore we use *strict* mock objects [Meszaros07].

How to record function calls

An important part of a mock object implementation is the recognition of the function calls the SUT makes on the mock object while a unit test runs.

Local classes: 2nd class citizens in the C++ world

Local classes are still not first-class citizens even in C++11. Declarations in local classes can only use type names, static and external variables, functions and enums from their enclosing scope. Access to automatic variables is therefore prohibited [ISO/IEC11]. Furthermore, they are also not allowed to have static and template members.

Beside the sequence and number of calls, we are also often interested in their argument values. We therefore have to store these facts to be able to later compare the calls with the users expectations.

We use an abstraction named **`call`** for this purpose which represents a call of a function. Its basic functionality is shown in Listing 4. A function call consists of the signature of the function and its argument values. Because we have to allow arguments of any type, we use a template parameter for the arguments in the constructor of **`call`**. Due to the fact that we do not want to restrict the number of arguments, we use a variadic template parameter pack.

The constructor of **`call`** uses the variadic template member function **`record`** to recursively process the arguments of the function call. **`record(Head const&, Tail const&)`** is used as the recursion step whereas **`record()`** handles the basic case of the recursion. Note the use of template parameter unpacking in the **`sizeof`** call to separate the argument values with commas and for the recursive call in record.

`call` uses a **`std::string`** object to store the function signature and the argument values. This is used to remember the values of any possible

```
// mockobjects.h
#include <sstream>
#include <vector>
struct call {
  template<typename ...Param>
  call(std::string const& funSig,
       Param const& ...params) {
       record(funSig, params ...);
  }
  template<typename Head, typename ...Tail>
  void record(Head const& head,
              Tail const& ...tail)  {
    std::ostringstream oss;
    toStream(oss, head);
    if (sizeof...(tail)) {
      oss « ",";
    }
    trace.append(oss.str());
    record(tail ...);
  }
  void record() { }
  std::string trace;
};
typedef std::vector<call> calls;
```

Listing 4

What are variadic templates?

Variadic templates – introduced with C++11 – basically address two limitations we have with the old C++ standard: the impossibility to instantiate class and function templates with arbitrary long parameter lists and to pass any number of arguments to a function in a type-safe manner. The ellipsis used on the left side of the parameter in variadic templates denote a so called *template parameter pack* which groups zero or more template arguments. The inverse action of *packing* is called unpacking and is applied when the `...` operator is used on the right side of a template or function call argument. Variadic templates are often used in combination with recursion which we also apply in this article.

argument type and to give the user as much information as possible when a comparison fails. Also note the **`typedef calls`** which we use to store the calls on an instance of a mock object class.

Requirements on function parameter types

To store the argument values in a string, we expect that types used for the function arguments implement a corresponding **`operator(ostream&, Type)`**. To prevent cryptic compiler errors if this is not the case, we use some template meta programming tricks taken from the Boost exception library. The interested reader might want to have a look at the file **`is_output_streamable.hpp`** in a recent Boost library version to see how this works. This is done in the function **`toStream`** which delegates the work of using the stream output operator in case it is defined and otherwise writing a message into the stream to inform the user about the missing operator.

```
template<typename T>
std::ostream& toStream(std::ostream& os,
                       T const& t) {
  selectbuiltinshiftif<T,
     isoutputstreamable<T>::value> out(os);
  return out(t);
}
```

Specifying expectations with initialiser lists

When unit testing our objects, we want to compare a list of function calls against our expectations. We use initialiser lists for specifying expectations the SUT has to fulfil. Note that C++ always allowed initialisation of plain old data (POD) types and arrays with initialiser lists, i.e., to give a list of arguments in curly brackets. But it was not possible in the old standard to use initialiser lists with regular (non-POD) classes. This has changed with C++11 where we are now able to instantiate regular classes with initialiser lists. This can be seen here with our **`calls`** vector.

```
calls expected = {
  {"foo(int i)", 42},
  {"bar(char c)", 'x'},
  {"foo(std::string s, double d)",
        "mockator", 3.1415}
};
```

In order to make comparisons work between the actual executed calls of the SUT on the mock object and our expectations, we have to provide an equality operator for **`call`**. **`operator==`** just delegates the work to the equality operator of **`std::string`** to compare the traced function call. To allow unit testing frameworks to print a string representation of the object under consideration if a comparison fails, we also provide a stream operator for **`call`**.

```
bool operator==(call const& lhs,
                call const& rhs) {
  return lhs.trace == rhs.trace;
}
std::ostream& operator«(std::ostream& os,
                        call const& c) {
  return os « c.trace;
}
```

Another important thing to explain is the function **`reserveNextCallId`** applied in Listing 3. This function is used to initialise the ID of the mock object and to add another call vector to the **`allCalls`** vector which collects all calls made on all instances of the mock object class. Its implementation is:

```
size_t reserveNextCallId
   (std::vector<calls> &allCalls) {
  size_t counter = allCalls.size();
  allCalls.pushback(calls{});
  return counter;
}
```

Reference implementation

Based on the mock object library discussed in this article, we have implemented Mockator. Mockator is a plug-in for the Eclipse C/C++ Development Tooling (CDT) platform including a header-only C++ based mock object library. The library also supports order-independent comparisons, the use of regular expressions in the expectations, nice string representations for easier comparisons of STL containers when used as function arguments and C++03 beside C++11.

Because common mock object libraries often lack good IDE support, we implemented a plug-in for Eclipse CDT that – beside its support for seams presented in the foregoing article –recognises missing member functions the SUT calls on the mock object and is able to generate them including the presented call registrations. It is able to generate code for both C++ standards and not only supports the common mock objects based on inhertiance, but also ones based on parametric polymorphism.

We recognised that it is often beneficial to just mock a single function instead of extracting an interface or a template parameter for classes. Therefore, we also implemented mocking of functions. Additionally, we provide various convenience functions to make working with mock objects easier like moving them to a namespace (useful if the unit test gets too big because of the mock object code and to share mock objects between unit tests), converting fake to mock objects, toggling the call recording on a member function level and recognising inconsistent expectations. The interested reader can download Mockator and give it a try. It is available as an alpha version under [Rüegg12]. ■

References

[Feathers04] *Working Effectively With Legacy Code*, Michael C. Feathers 2004

[ISO/IEC11] Working Draft, Standard for Programming Language C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, February 2011

[Meszaros07] Gerard Meszaros, *Unit Test Patterns: Refactoring Test Code*, Addison-Wesley 2007

[Rüegg12] Michael Rüegg, 'Mockator', available from http://www.mockator.com, 2012

Large Objects and Iterator Blocks

Arrays can cause memory issues in .Net. Frances Buontempo shows how iterator blocks can help to relieve the pressure.

C# is a garbage collected language. This allows you to create new objects at will, and the garbage collector will clear them up as and when needed. Despite this, out of memory errors seem to be frighteningly common in some large scale C# applications I have come across. It isn't just me: Stackoverflow [SO] gives nearly 3,000 questions related to C# and 'out of memory'. How is this possible, given the hype about automatic memory management through the garbage collector?

Previous articles [Overload63] have dealt with the IDISPOSABLE pattern, with reference to avoiding leaks. This article will instead focus on objects that are likely to make it to the large object heap and therefore probably persist for the lifetime of an application. Let us begin with a brief overview of garbage collection.

Garbage

Garbage collection has a long history. Wikipedia claims 'Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp' [GC]. It is an active area of research, used by a variety of languages including python, Java and C# [REJ]. Garbage collection can be implemented in a variety of ways, for example reference counting with cycle detection versus generational, blocking versus concurrent, and many other variations.

.Net uses a generational garbage collector, which 'moves' or compacts objects after a collection. Every time a new object is created, it is placed in generation zero. When the garbage collector runs, it drops unused objects from generation zero, queuing up finalizers. The surviving objects are then compacted, unless they have been pinned. Anything that survives the next run gets promoted to generation one. The same happens from time to time with generation one and surviving objects get promoted to generation two. Less frequently, generation two is inspected along with the large object heap. These objects might well last for the lifetime of an application.

This begs the question, 'What does 'large' mean?'

> Objects that are greater than approximately 85,000 bytes are treated as large objects by the garbage collector and are directly allocated in a special heap; they are not promoted through the generations. [LOH]

This decision appears to be a performance decision [LOH-MS]. For example, compacting large objects takes longer than compacting small objects. As noted, large objects are only collected when a generation two collection happens, so they could stay around for a very long time, thereby using up a large amount of memory. This can clearly cause problems.

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a professional programmer for over 12 years. She can be contacted at frances.buontempo@gmail.com.

Avoiding arrays

How could an object of approximately 85,000 bytes get created? Though many applications end up with gigantic strings, for example reading a whole file into memory or working with huge amounts of xml, we will focus on numbers rather than strings. Or at least arrays, which frequently crop up in mathematical contexts and contain doubles, for example in a MonteCarlo simulation with, say 100,000 paths, immediately providing over 85,000 bytes. The internet also suggests that arrays of 1,000 or more doubles get placed on the large object heap [LOH-Doubles], again for performance reasons: 'The LOH is aligned to 8 bytes so access to 'big' arrays is much faster if it is allocated on the LOH...the best tradeoff is when the array is bigger than 1000 elements.'

It is hard to imagine why you would ever need all the numbers in the array to be in memory at once. Frequently, an average, maximum or minimum of these numbers will be required. This can be calculated if functions return an **`IEnumerable<double>`** rather than a **`double[]`**, in other words, if we provide an iterator block instead. These have been around since .Net 2 and a clear and thorough explanation is given in [Skeet]. Put simply, they are 'syntactic sugar' which causes the compiler to build a state machine, allowing each request for the next item to be lazily evaluated.

Iterator blocks will work for other objects besides doubles, but without loss of generality consider the following fictitious function.

```
public static double[] Value()
{
  double[] res = new double[10000];
  for (int i = 0; i < 10000; ++i) res[i] = 42.0;
  return res;
}
```

The calling code is likely to take the form

```
var v = Value();
foreach (double d in v)
{
//do something
}
```

The following simple change to the function will reduce the lines of code, which is always a good thing, and reduce the memory it uses, without changing the client code.

```
public static IEnumerable<double> Value()
{
  for (int i = 0; i < 10000; ++i)
      yield return 42.0;
}
```

This no longer allocates an array. Since the calling code simply iterates over the items, the iterator block can now yield the required numbers one at a time, using less memory. Significantly, this will no longer shove an array on the large object heap, potentially leaving it hanging around for the lifetime of the application.

Clues can be provided by the Performance counters which will give further details about memory usage in each generation and on the large object heap

Prove it

A high level view of total memory usage is provided by the System.GC object.

```
static public void ShowMemory(string explanation)
{
  long memory = GC.GetTotalMemory(true);
  Console.WriteLine("{0,]30} {1}",
  explanation + ":", memory);
}
```

Calling our first value function which allocates a large array of doubles gives 635056 bytes, whereas the second version gives 555088. Though this high level view doesn't say exactly where the memory is, it does show less is being used. In order to see exactly which objects are taking up how much space, we'd need to use a profiler, such as Microsoft's CLRProfiler [CLRProfiler4], or use SOS in Windebug, which is beyond the scope of this article. Clues can be provided by the Performance counters which will give further details about memory usage in each generation and on the large object heap. Sample code is shown in by the `ShowGensAndLOHMemory` function. Note that some counters update at the end of a garbage collection, not at each allocation [LOH], so you need to provoke a garbage collection in order to get accurate counts back. The `ShowMemory` function will do this, since we send `true` to the GC's `GetTotalMemory` function, which forces a full collection (Listing 1).

If this is called on the `Value` functions above we see a reduction in large object heap memory usage. This proves we have used less overall memory, and specifically less of the large object heap. See Table 1 for details.

A final example

Suppose we wish to do something a bit more complicated, such as bale out without returning values if a condition is met (Listing 2).

It is likely the calling code will be very similar to the initial simpler example, though the function returning the numbers is now a little more complicated. This can still be changed to use iterator blocks, most likely without changing the calling code. (Listing 3)

In this case, we yield break when there is nothing to return. It does not have to be at the start of the iterator block function, for example it could happen if a condition was met within the main loop instead. In this example, data

	Arrays	Iterator Block
(loh) memory	126248	46216
(Gen0) memory	4194300	4194300
(Gen1) memory	12	12
(Gen2) memory	520004	520052

Table 1

```
static public void ShowGensAndLOHMemory()
{
  PerformanceCounter loh =
     new PerformanceCounter(".NET CLR Memory",
     "Large Object Heap size",
     Process.GetCurrentProcess().ProcessName,
     true);
  PerformanceCounter perfGen0Heap =
     new PerformanceCounter(".NET CLR Memory",
     "Gen 0 heap size",
     Process.GetCurrentProcess().ProcessName,
     true);
  PerformanceCounter perfGen1Heap =
     new PerformanceCounter(".NET CLR Memory",
     "Gen 1 heap size",
     Process.GetCurrentProcess().ProcessName,
     true);
  PerformanceCounter perfGen2Heap =
     new PerformanceCounter(".NET CLR Memory",
     "Gen 2 heap size",
     Process.GetCurrentProcess().ProcessName,
     true);

  Console.WriteLine("(loh) memory: {0}",
     loh.NextValue());
  Console.WriteLine("(Gen0) memory: {0}",
     perfGen0Heap.NextValue());
  Console.WriteLine("(Gen1) memory: {0}",
     perfGen1Heap.NextValue());
  Console.WriteLine("(Gen2) memory: {0}",
     perfGen2Heap.NextValue());
}
```

Listing 1

```
public static double[] ValueAtTime(double t)
{
  Dictionary<double, double> data =
     new Dictionary<double, double> {{-1.0,10.0},
     {0.0, 20.0}, {1.0, 30.0}, {2.0, 40.0}};
  var dates = data.Where(a => a.Key < t);
  if (!dates.Any())
    return new double[0];
  double amount =
     dates.OrderBy(a => a.Key).Last().Value;
  double[] res = new double[10000];
  for (int i = 0; i < 10000; ++i)
      res[i] = amount;
  return res;
}
```

Listing 2

It is surprisingly easy to **run out of memory** in .Net, but there are some simple things you can do to **reduce memory usage**

```
public static IEnumerable<double>
   ValueAtTime (double t)
{
  Dictionary<double, double> data =
     new Dictionary<double, double> {
       { -1.0, 10.0 }, { 0.0, 20.0 },
       { 1.0, 30.0 }, { 2.0, 40.0 } };
  var dates = data.Where(a => a.Key < t);
  if (!dates.Any())
     yield break;
  double amount =
     dates.OrderBy(a => a.Key).Last().Value;
  for (int i = 0; i < 10000; ++i)
      yield return amount;
}
```

Listing 3

is small, but a more realistic example with a huge amount of data will reduce the memory, because we don't create an array upfront.

.Net 4 allows us to write this with even fewer lines of code, using **`Enumerable`**. (Listing 4)

Caveats

We have seen that iterator blocks can help with memory issues, however a couple of things need to be borne in mind. You need to take care where you place yield statements. In particular, you can't yield from a **`try`** block with a **`catch`** block, or in a **`catch`** block or a **`finally`** block. In addition, the client code may never consume the whole iteration. This means if it contains a **`finally`** block, either explicitly or through a **`using`** statement, that may never be executed, unless the iterator block is wrapped in a **`using`** statement. Locks and threading can be dangerous too. See [Skeet] for more details.

```
public static IEnumerable<double>
   ValueAtTime (double t)
{
  Dictionary<double, double> data =
     new Dictionary<double, double> {
       { -1.0, 10.0 }, { 0.0, 20.0 },
       { 1.0, 30.0 }, { 2.0, 40.0 } };
  var dates = data.Where(a => a.Key < t);
  if (!dates.Any())
     return Enumerable.Empty< double >();
  return Enumerable.Repeat
     ( dates.OrderBy(a => a.Key).Last().Value,
       10000 );
}
```

Listing 4

Conclusion

It is surprisingly easy to run out of memory in .Net, but there are some simple things you can do to reduce memory usage. This article has shown how iterator blocks can reduce memory footprint without changing the client code. Significantly, if an object is large enough to end up on the LOH, swapping to iterator blocks where possible could stop your application falling over due to a **`System.OutOfMemory`** exception. There are many other ways to reduce memory that we have not considered here, such as using IDISPOSABLE, interning strings and remembering to state a capacity of a **`List<T>`** on construction, since calling **`Add`** will possibly do a reallocation, dumping the previous list, which could potentially be on the large object heap already, giving you two large objects with the price of one. ■

References

[CLRProfiler4] http://www.microsoft.com/en-us/download/details.aspx?id=16273

[GC] http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[LOH] http://msdn.microsoft.com/en-us/library/x2tyfybc(v=vs.90).aspx

[LOH-Doubles] https://connect.microsoft.com/VisualStudio/feedback/details/266330/large-object-heap-loh-does-not-behave-as-expected-for-double-array-placement

[LOH-MS] http://msdn.microsoft.com/en-us/magazine/cc534993.aspx

[Overload63] 'Garbage Collection and Object Lifetime' Ric Parkin, *Overload* #63, Oct 2004.

[REJ] http://www.cs.kent.ac.uk/people/staff/rej/gc.html

[Skeet] http://csharpindepth.com/Articles/Chapter6/IteratorBlockImplementation.aspx

[SO] http://stackoverflow.com/search?q=C%23+out+of+memory+