Performance Optimisation Part 1

by Bob Swart

Delphi is based on components. This means we have a new way of optimising our applications. Traditionally, we've used a tool like Turbo Profiler to find the efficiency bottle-necks in our application: a *top down* approach. With Delphi, the application consists of components that interact together, so we can use a *bottom up* approach, where we leave the application itself intact, but focus on optimising the underlying components. Without changing one line of code in the application itself, we can increase performance by installing faster versions of the used components.

Performance Optimisation

Hopefully, it is common knowledge that many applications contain small parts of code which are responsible for major parts of the execution time. This three-part series of articles will focus on optimisation techniques for program execution speed and program size. If we can write faster, smaller programs, we can gain that little extra to discriminate us and our applications from the rest!

In order to do so, we must first divide the process of performance optimisation into six major phases:

1. We need to customise our Delphi compiler and linker options for maximum efficiency.

2. We must identify bottle-necks, using Turbo Profiler or another profiling or timing tool. We will examine tools to measure the number of times a statement block is executed, and several methods of timing these statement blocks. Statement blocks may be statements, but also macros, procedures, functions, units and whole programs. We can find the performance bottle-necks using these techniques. It will turn out that even small improvements in these bottle-necks will often have more effect than big improvements in non-bottle-neck areas.

➤ *Figure 1 Compiler and linker options*

3. We should examine the data structures and algorithms of the bottle-necks found in step 2 and try to find more efficient equivalents. We will see that the largest performance improvements can come from changing algorithms and data structures. A more specific solution to a problem might be less flexible, but can often be an order of magnitude faster!

4. If step 3 fails, we should examine the source code and look for more efficient VCL methods, properties or language constructs to use in the data structures and algorithms.

5. If step 4 fails, we should examine a mixed ObjectPascal/Assembly listing and then try to rewrite the algorithms, routines or statements in Assembler, using external linked assembler object code, BASM or InLine macros.

6. If the code is now as fast as possible, we can try to minimise its size, looking at code size, data size, stack space, heap space and overall executable size. Sometimes, we might even prefer optimisations for size rather than speed.

A step can fail in two ways: firstly if it simply cannot be done and secondly if it doesn't deliver the required performance result.

First Things First

Before we even start to think about performance optimisation, it is time to check out the compiler options: although they have less impact than a bad algorithm, they can seriously slow down your code and expand your total code size. My personal project options are shown in Figure 1 (note the checked "Default", which means that these options are now default for every new project).

In the IDE we can easily obtain the current settings of the compiler directive by typing `Ctrl+O-O` in edit-mode. The current settings will then be inserted at the beginning of the source as compiler directives. We can change them to our liking and also make sure a recompile on another user's system (with perhaps other general compiler options set) still yields the same results. You can also view the options in the project's .OPT file.

Finding Bottle-Necks

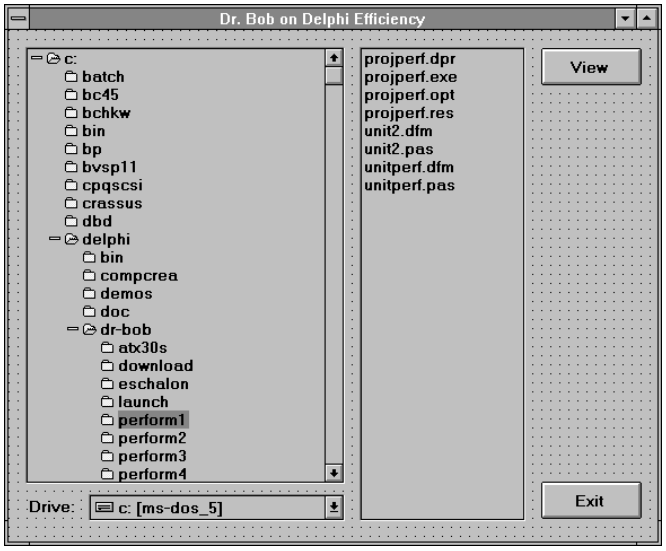
Never think you know where performance bottle-necks are in your code. Always measure with some kind of tool. Use Turbo Profiler, or one of the other profiling techniques we will examine here, but never think you know before you measure. We could accidentally be right, but when using profiling tools and techniques we'll be sure to find all the bottle-necks!

When using Turbo Profiler, we need to set the following compiler and linker options: {$D+,L+,Y+} (in the IDE, see Figure 1, check `Debug information`, `Local symbols` and `Symbol info`) and include the TDW debug info in the executable (Turbo Profiler and Turbo Debugger are actually internally similar).

```
program CrtApp;
uses WinCrt, WinProcs;
var StartTick: LongInt;
    Reps: LongInt;
begin
  Reps := 0;
  StartTick := GetTickCount;
  while StartTick = GetTickCount
    do {wait for end of Tick};
  StartTick := GetTickCount;
  repeat
    { call your test function }
    Inc(Reps);
  until StartTick <>
    GetTickCount;
  writeln(Reps);
end.
```

➤ *Listing 1*

➤ *Figure 2*



Unfortunately, Delphi does not currently ship with a Profiler and the Turbo Profiler 2.2 with Borland Pascal is unable to read the debug format of Delphi. You'll need Turbo Profiler 4.5, from Borland C++ 4.5.

Using Turbo Profiler, we can apply a top-down search for the bottle-necks. First we must identify the component which takes most of the execution time. Within this component, we have to identify the methods that take up most time. Finally, we can profile all lines within this time-eating method.

Alternatively, or if you don't have Turbo Profiler, we can implement our own timing methods using either the Windows System Clock or the Reps Timer technique.

The `GetTickCount` API gives us the number of milliseconds since Windows was started. The accuracy of this API, however, is only 55 ms, as it is updated 18.2 times each second (just like the real mode system clock). Hence, we should use `GetTickCount` only for very rough measurements.

From the `MMSYSTEM.DLL` we can use the function `timeGetTime` (at index 607), which returns a `LongInt` and is accurate to one millisecond!

These two APIs only show the total time that has elapsed during a certain operation, without taking into account the time that *other applications* consumed. For better measuring, we can use the `TOOLHELP.DLL` function `TimerCount` which gives the time spent in *our virtual machine only*.

Sometimes a given routine or piece of code is just too fast to be timed by the above System Clock technique. Can this piece of code still be a bottle-neck? Yes it can, for example when called thousands or millions of times. In these cases, like a tight loop, even the smallest change counts. To measure these kind of fast routines or statements we can't use the system clock timer, because its resolution isn't good enough. We need the Reps Timer: a small routine which doesn't count the execution time of a piece of code, but just counts the number of times it can be executed in one (or more) clock ticks. We can use this count to compare several blocks of code against each other. See Listing 1 for an example.

Now we can measure the number of executions of a function per clock-tick. Note that we have some overhead from incrementing `Reps` every time in the loop, and from checking whether `StartTick` is already equal to `GetTickCount`. As a consequence, a piece of code that does twice as many `Reps` is probably *more* than twice as fast.

Algorithms & Data Structures

Once we've found the bottle-necks in our application, the next thing we must do is check the algorithm or data structures. Often, by overcoming inefficiency here we can speed up the application by an order of magnitude. Examples of more efficient algorithms to replace an inefficient one are: a Binary Search to replace a Linear Search, a Quick Sort to replace a Bubble Sort, or a BTree to replace a linear linked list.

A Real-Life Example

What better way to illustrate this point than by building an application? Drop a `DriveComboBox`, `DirectoryOutliner`, a `FileListBox` and a few `Buttons` on a form, like Figure 2. Connect the `DriveComboBox` with the `DirectoryOutLiner` and the `DirectoryOutLiner` with the `FileListBox` as shown in Listing 2.

If we run the example program, we see that it takes a noticeable time to load the directory outliner. If we change to a directory with many subdirectories and open it up for the first time, we may have to wait more than 10 seconds before the outliner is re-drawn. Let's try to decrease this time to less than 1 second!

Since we have already found out that there is a possible bottle-neck in the `TDirectoryOutline` component, I fired up Turbo Profiler 4.5 and marked each method of `TDirectoryOutline` as an active area for the profiler. After running the program for a short while, I found that the `Expand`, `Click` and `BuildOneLevel` methods were the three greatest time-consumers.

There is an important danger we need to be aware of when using Turbo Profiler on a Windows application. Are we measuring internal calculations, or are we measuring end-user reaction times and input speed to the user interface? The procedure `Click` is just an user input event handler. Also, within `Expand`, the call to `inherited Expand` (Listing 3) is just a call to paint the outliner on the screen.

So, going back to Turbo Profiler, I narrowed my search to examine the time spent in *each line* of the `Expand` and `BuildOneLevel` procedures. The two major time eaters are part of a linear search algorithm in procedure `BuildOneLevel` used to place new nodes in alphabetic order under their parent: see the red lines in Listing 4.

If you check out the source code of `TDirectoryOutline` (in `\DELPHI\SOURCE\SAMPLES`), you can find the bottle-neck. It uses an very slow linear insertion method to add new subdirectory entries to a directory in the outliner. We could disable the sorting, but a better solution is to use a faster algorithm!

Bottom-Up Efficiency

In order to enhance the performance of `TDirectoryOutline`, we have to modify DIROUTLN.PAS. (I made a copy of this file and put it in my project directory; now, the enhancements I make in this "local" DIROUTLN.PAS will affect the whole project, since the "local" version overrides COMPLIB.DCL).

I call this feature "bottom-up efficiency", as it enables us to write more efficient versions of components and enhance efficiency by simply recompiling the applications that use these components! Specifically, if we use a better algorithm (like binary insertion) for `TDirectoryOutline`, we can get a *tenfold increase in speed*. Now, a directory with 100 subdirectories takes under a second to load.

The binary insertion code I used for the tenfold increase in speed is placed in a local routine of procedure `BuildOneLevel` as shown in Listing 5. Don't forget to replace the two bottle-neck lines in `BuildOneLevel` with a single call to `FindIndex`, like so:

```
TempChild :=
  FindIndex(RootNode,
  SearchRec.Name); { Dr. Bob }
```

Conclusions

In this first part of the series we've seen a structured performance optimisation process, where we (top-down) break down the application and optimise step by step.

```
procedure TMainForm.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryOutLine1.Drive := DriveComboBox1.Drive;
end;
procedure TMainForm.DirectoryOutline1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryOutline1.Directory;
end;
```

➤ *Listing 2*

```
procedure TDirectoryOutline.Expand(Index: Longint);
begin
  if Items[Index].Data = nil then
    BuildOneLevel(Index);
  inherited Expand(Index);
end;
```

➤ *Listing 3*

```
0.0002 151    if RootNode.HasItems then  {if has children, must alphabetise}
              begin
0.1655 146      TempChild := RootNode.GetFirstChild;
11.684 5326     while (TempChild <> InvalidIndex) and
                      (Items[TempChild].Text < SearchRec.Name) do
12.076 5180       TempChild := RootNode.GetNextChild(TempChild);
0.1105 146      if TempChild <> InvalidIndex then
0.6892 132        NewChild := Insert(TempChild, SearchRec.Name)
0.0115 14       else NewChild := Add(RootNode.GetLastChild, SearchRec.Name);
0.0001 146    end
```

➤ *Listing 4*

```
function FindIndex(RootNode: TOutLineNode; SearchName: TFileName): LongInt;
var FirstChild,LastChild,TempChild: LongInt;
begin
  FirstChild := RootNode.GetFirstChild;
  if (FirstChild = InvalidIndex) or
     (SearchName <= Items[FirstChild].Text) then FindIndex := FirstChild
  else begin
    LastChild := RootNode.GetLastChild;
    if (SearchName >= Items[LastChild].Text) then FindIndex := InvalidIndex
    else begin
      repeat
        TempChild := (FirstChild + LastChild) div 2; { binary search }
        if (TempChild = FirstChild) then Inc(TempChild);
        if (SearchName > Items[TempChild].Text) then FirstChild := TempChild
          else LastChild := TempChild
      until FirstChild >= (LastChild-1);
      FindIndex := LastChild
    end
  end
end {FindIndex};
```

➤ *Listing 5*

We've also seen special bottom-up optimisation, a feature introduced by the re-usable component nature of Delphi. The source code for the examples is on this issue's disk. In the next part we'll examine how to add new functionality and efficient DLLs to existing applications.

Bob Swart is a professional 16- and 32-bit software developer using Borland Pascal, C++ and Delphi. In his spare time, he likes to watch video tapes of Star Trek Voyager with his 1.5 year old son Erik Mark Pascal. Email Bob on 100434.2072@compuserve.com

Acknowledgements: *the first two parts of this series are based on my talk in session DL390 (Delphi Performance Optimisation) at the 6th Annual Borland Developers Conference, August 6-9 1995, in San Diego, USA.*