Speeding Up A Mathematical Expression Parser

Optimisation by generating machine code at runtime

by Hallvard Vassbotn

One of the more useful capabilities of a flexible application is runtime interpretation of scripts, formulae or other textual information. As users of a compiled language like Object Pascal, most of us cringe at the idea of spending valuable execution time spinning around in an interpreter loop, reading raw text or at best p-code terms and dispatching them.

If the system has the right design, we can often improve the runtime performance dramatically by pre-compiling a corresponding stream of sequential machine code instructions. This is particularly true in situations where the interpreted code will be called frequently, as in the case of plotting the chart of a user-supplied function. In this article I will show one example of how such an optimisation can be done.

Confessions Of A Surfer

Quite some time ago I was surfing around the web, looking for interesting programming sites in general and Delphi sites in particular. I find it a little disappointing that many sites contain not much more than links to other sites. However, one of the sites I visited was Agner Fog's[1]. His Pentium assembly optimisation manual has become a de facto reference work, more detailed and more correct than Intel's own documentation. It's highly recommended reading if you are into optimisations and assembly programming, or just want to know more about what happens inside the processor.

Pentium Secrets

At the time, I focused on the key interesting points that I did not know much about. I particularly noted that the Pentium II and later processors have amazingly improved branch prediction logic, a very high misprediction penalty, and that indirect calls are always predicted to go to the same target. The following quotes document this in more detail:

'22.2.2 Misprediction penalty (PMMX, PPro, PII and PIII)

'In the PPro, PII and PIII, the misprediction penalty is very high due to the long pipeline. A misprediction usually costs between 10 and 20 clock cycles. It is therefore very important to be aware of poorly predictable branches when running on PPro, PII and PIII.

'22.2.3 Pattern recognition for conditional jumps (PMMX, PPro, PII and PIII)

'These processors have an advanced pattern recognition mechanism which will correctly predict a branch instruction which, for example, is taken every fourth time and falls through the other three times. In fact, they can predict any repetitive pattern of jumps and nojumps with a period of up to five, and many patterns with higher periods...

'22.2.9 Indirect jumps and calls (PMMX, PPro, PII and PIII)

'There is no pattern recognition for indirect jumps and calls, and the BTB can remember no more than one target for an indirect jump. It is simply predicted to go to the same target as it did last time.'

This implies that you will have many costly processor stalls when calling code through pointers if the target address isn't always the same. This is just what happens when you call virtual methods of different object types. The addresses of the virtual methods are stored in the object instance's VMT table. In two objects of different class types, that both override a virtual method, the VMT will point to different methods. So, when it is time to call the virtual method, the processor will stall every time you switch from one object type to another. This typically happens when you call a virtual method for all the objects in a list or collection of some sort. Isn't it ironic that today's modern processors are not optimised to execute object-oriented code ☺?

The Fastest Freeware Parser In The World

At about the same time, I took a closer look at Stefan Hoffmeister's site[2], downloading (amongst other interesting material) his version of the Parser10 library.

This component parses mathematical expressions at runtime, evaluating them fairly efficiently. The readme file claimed a performance only 40% to 80% slower than the corresponding native compiled Delphi code, and that made it the fastest parser on the freeware market. The component has been successively developed by Renate Schaaf[3], Alin Flaidăr[4] and Stefan Hoffmeister.

➤ *Listing 1*

```
POperation = ^TOperation;
TMathProcedure = procedure(AnOperation: POperation);
TOperation = record
  Arg1: PParserFloat;
  Arg2: PParserFloat;
  Dest: PParserFloat;
  NextOperation: POperation;
  MathProc: TMathProcedure;
  Token: TToken;
end;
```

Upon examining the code, I found that it could accomplish this impressive performance by converting the text expression into a linked list of operation records. Each record contains a pointer to a pre-coded routine that performs one mathematical operation (such as addition, multiplication, sine, etc). See Listing 1.

Evaluating the expression is then as simple as following the linked list in a loop, calling through the procedure pointer in each iteration. See Listing 2.

As I was reading through this code, I immediately got the famous light-bulb effect in my brain. It was very clear to me that this loop would be severely affected by the misprediction penalties described by Fog. More often than not, the next operation would be different to the previous one. So, for each iteration, the processor would stall for 10 to 20 cycles before moving on. It should therefore be possible to improve the performance of this code by avoiding these stalls.

I will come back to this later. First, let us first look a little more closely at the interface of this parser component and how we would typically use it.

The Vanilla Version

The original version of Parser10 performed very well and was very flexible and nicely designed. It had a number of useful mathematical functions built-in and you could easily add your own functions and variables. You can see the public interface of the component in Listing 3.

`TCustomParser` is the base class that contains the logic and adds support for the mandatory operators (+, -, /, * and so on). `TParser` adds basic mathematical functions (such as `SIN`, `TAN` and `SQRT`) and some predefined variables (`A`, `B`, `C`, etc) for convenience. Table 1 lists the supported constants, operators and functions.

In basic use, you set the `Expression` property to the formula you want to evaluate. Then you can set the value of one or more of the built-in variables by setting the corresponding property (`A`, `B`, `X`, etc). Finally, you evaluate the expression by reading the `Value` property.

You can use the `ClearFunction` method to remove the definition of an existing function or one of the two `AddFunction` methods to add new ones. The mathematical functions can take one or two input parameters (for instance, `SIN` takes one parameter, while `MAX` takes two) and they always produce one output parameter. These functions are implemented as procedures (yeah, I know that sounds a bit corny) that have a single pointer to `TOperation` parameter. Listing 4 shows how the built-in functions `SIN` and `MAX` have been implemented. If you want to add your own functions, follow the same pattern.

➤ *Table 1*

Constants	PI
Operators	+ , - , * , / , ^ , MOD, DIV
Functions	COS, SIN, SINH, COSH, TAN, COTAN, ARCTAN, ARG, EXP, LN, LOG10, LOG2, LOGN, SQRT, SQR, POWER, INTPOWER, MIN, MAX, ABS, TRUNC, INT, CEIL, FLOOR, HEAV, SIGN, ZERO, PH, RND, RANDOM

➤ *Listing 2*

```
function TCustomParser.GetValue: Extended;
var
  LastOP: POperation;
begin
  if FStartOperationList <> nil then begin
    LastOP := FStartOperationList;
    while LastOP^.NextOperation <> nil do begin
      with LastOP^ do begin
        MathProc(LastOP);
        LastOP := NextOperation;
      end;
    end;
    LastOP^.MathProc(LastOP);
    Result := LastOP^.Dest^;
  end else
    Result := 0;
end;
```

➤ *Listing 3*

```
TCustomParser = class(TComponent)
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  { Function support }
  procedure AddFunctionOneParam(const AFunctionName: string;
    const Func: TMathProcedure);
  procedure AddFunctionTwoParam(const AFunctionName: string;
    const Func: TMathProcedure);
  procedure ClearFunctions;
  procedure ClearFunction(const AFunctionName: string);
  { Variable support }
  procedure ClearVariables;
  procedure ClearVariable(const AVarName: string);
  function  VariableExists(const AVarName: string): boolean;
  function SetVariable(VarName: string; const Value:
    Extended): PParserFloat;
  property Variable[const VarName: string]: Extended
    read GetVariable write SetVar;
  { Error handling }
  property ParserError: boolean read FParserError;
published
  { To evaluate an expression simply read the
     Value property. }
  property Value: Extended read GetValue write SetValue
    stored False;
  property Expression: string read FExpression
    write SetExpression;
  property PascalNumberformat: boolean
    read FPascalNumberformat write FPascalNumberformat
    default True;
  property OnParserError: TParserExceptionEvent
    read FOnParserError write FOnParserError;
end;
TParser = class(TCustomParser)
public
  constructor Create(AOwner: TComponent); override;
published
  { predefined variable properties }
  property A: ParserFloat read FA write FA;
  property B: ParserFloat read FB write FB;
  property C: ParserFloat read FC write FC;
  property D: ParserFloat read FD write FD;
  property E: ParserFloat read FE write FE;
  property T: ParserFloat read FT write FT;
  property X: ParserFloat read FX write FX;
  property Y: ParserFloat read FY write FY;
end;
```

Arg1 is the first parameter, Arg2 is the (optional) second parameter, while Dest is where you should put the result of the mathematical operation. All three parameters are pointers to floating point variables (doubles). With this design, the Dest pointer of the first operation can directly update the Arg1 parameter of the next operation. The two fields both point to the same double variable. This clever trick allows quick execution with no copying of the result values into the parameters of the next operation. The result and parameter variables are simply aliases for each other. The complex logic in the P10Build unit is responsible for setting up the linked Operation records and to ensure that this aliasing is correct.

On this month's disk is a simple demonstration project (cunningly named SimpleDemo.dpr) that shows how to perform basic operations, such as defining new functions, defining new variables, setting the value of a variable, getting the value of a variable, setting the expression and also evaluating the expression. You can see the code for this in Listing 5.

Black Magic

The ParseFunction routine in the P10Build unit takes the original string expression (typically entered by the end-user at runtime) along with string lists representing the variables and the one- and two-parameter functions and, after chugging along on this for a while, it returns the linked list of operations, ready to be executed.

I trimmed this unit by removing some unused and buggy code to dynamically handle deeply nested expressions. I also simplified some conditional code that dealt with long versus short strings. I might have inadvertently made the code incompatible with Delphi 1, but at least the code is a little cleaner (and we are only generating 32-bit code anyway).

Still, this monstrous routine weighs in at about 1,500 lines, including a number of nested, forwarded, routines. The logic here could no doubt be improved a little; rewriting it as a class, with the nested routines as methods, wouldn't hurt either. In this respect, the code for the parser presented by Chris McNeil[5] is probably a better model.

However, this article is not about making the parsing of the expression string into terms as efficient and elegant as possible. It is about optimising the execution of those terms. For now, we will treat this piece of code as a black box, not bothering to dive into its dark inner details.

Limitations

The parser uses a few brute-force techniques (such as searching for substrings using Pos), so there are a few usage limitations. The most severe and noticeable limitation is that you cannot embed spaces in the formula expression. The number of nested brackets is limited to 20 (controlled by the maxBracketLevels constant) and the maximum number of terms within each bracket (I think) is limited to 50 (controlled by the maxLevelWidth constant).

Be sure to also read the original Parser10 documentation and readme file. I have not been able to update the help files, but most of the information in it is still accurate.

Speeding It Up

OK, now we have examined how the parser component works and how to use it. Let's start the fun and see how we can speed it up

➤ *Listing 4*

```
procedure _sin(AnOp: POperation); far;
begin
  with AnOp^ do
    dest^ := sin(arg1^);
end;
procedure _max(AnOp: POperation); far;
begin
  with AnOp^ do
    if arg1^ < arg2^ then
      dest^ := arg2^
    else
      dest^ := arg1^;
end;
```

➤ *Listing 5*

```
procedure TSimpleDemoForm.CalcBtnClick(Sender: TObject);
begin
  // Setting the expression
  Parser.Expression := ExpressionEdit.Text;
  // Evaluating the expression
  ResultEdit.Text := FloatToStr(Parser.Value);
end;
procedure TestOneParam(AnOp: POperation);
begin
  with AnOp^ do
    dest^ := arg1^*3;
end;
procedure TestTwoParam(AnOp: POperation);
begin
  with AnOp^ do
    dest^ := arg1^ + arg2^*2;
end;
procedure TSimpleDemoForm.DefineFuncsBtnClick(Sender:
  TObject);
begin
  // Adding functions
  Parser.AddFunctionOneParam('OneParam', TestOneParam);
  Parser.AddFunctionTwoParam('TwoParam', TestTwoParam);
end;
procedure TSimpleDemoForm.RemoveFuncsBtnClick(Sender:
  TObject);
begin
  // Removing functions
  Parser.ClearFunction('OneParam');
  Parser.ClearFunction('TwoParam');
end;
procedure TSimpleDemoForm.DefineVarBtnClick(Sender:
  TObject);
var
  MyVar2: PParserFloat;
begin
  // Setting variables - slow
  Parser.Variable['MyVar1'] := 3.14;
  MyVar2 := Parser.SetVariable('MyVar2', 0);
  // Setting variables - fast
  MyVar2^ := 50;
  // Setting built-in variables, fast
  Parser.A := 60;
  // Getting variables - slow
  ShowMessage('Value of MyVar1 = ' +
    FloatToStr(Parser.Variable['MyVar1']));
  // Getting variables - fast
  ShowMessage('Value of MyVar2 = ' + FloatToStr(MyVar2^));
  // Getting built-in variables, fast
  ShowMessage('Value of A = ' + FloatToStr(Parser.A));
end;
```

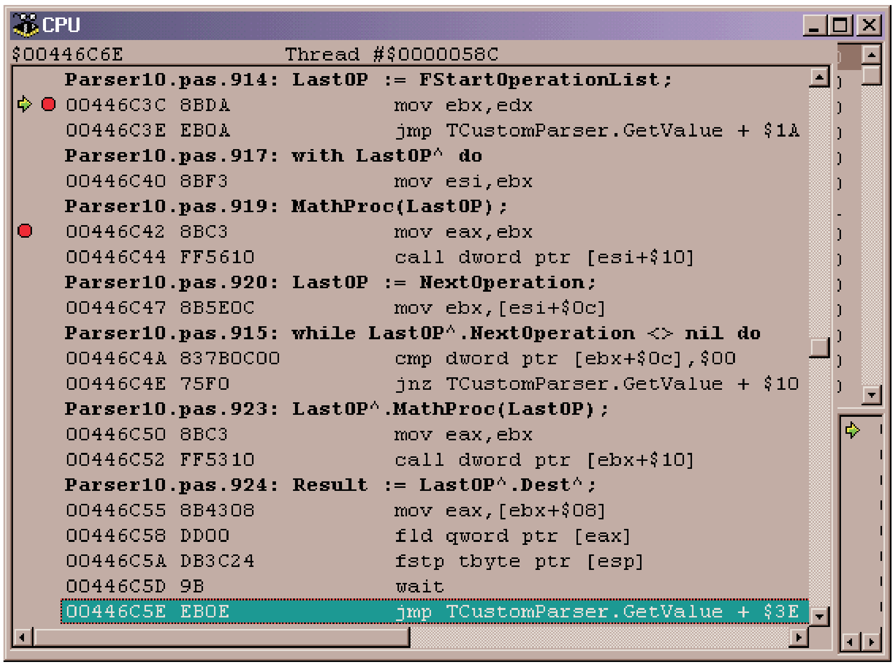
even more. The evaluation of the expression after all the variable values have been set basically boils down to the simple loop we looked at in Listing 2. Simply put, the expressions have been put into reverse Polish notation and converted to a linked list of operation records, as defined in Listing 1.

In this record, the `MathProc` field is a pointer to a procedure that will perform the desired calculation. This loop executes fairly fast; however, there are a couple of problems with it. First of all, it is a very small loop, so the CPU spends a fair amount of time just jumping around. Secondly, and much worse, the branch targeting prediction of the Pentium II and later processors will be seriously handicapped by this code. The problem is that in most cases the operation pointers will point to different procedures for each run through the loop. As Fog explains, the Pentium processors have branch target prediction logic that is tied to the address the call is being made from. If the code calls the same target twice in a row from the same address, execution is improved. If not, there will be a mis-prediction and the processor must be stalled and execution pipes must be flushed.

To kill both birds with one stone, the code would improve significantly if we somehow could unroll the loop: something like the example shown in Listing 6.

The problem is that we don't know at compile-time how many levels we should unroll. We could check for a `nil` condition each time, but that would slow us down again, and we would have to include a large quantity of identical code.

A better solution would be to play the compiler. The list of operation records will be our 'source code' and we should generate machine code that mimics the unrolled loop in Listing 7. The first step is to examine the code that the compiler generates. Figure 1 shows the CPU view of the code generated for Listing 6.



```
$00446C6E                 Thread #$0000058C
    Parser10.pas.914: LastOP := FStartOperationList;
    00446C3C 8BDA              mov ebx,edx
    00446C3E EB0A              jmp TCustomParser.GetValue + $1A
    Parser10.pas.917: with LastOP^ do
    00446C40 8BF3              mov esi,ebx
    Parser10.pas.919: MathProc(LastOP);
    00446C42 8BC3              mov eax,ebx
    00446C44 FF5610            call dword ptr [esi+$10]
    Parser10.pas.920: LastOP := NextOperation;
    00446C47 8B5E0C            mov ebx,[esi+$0c]
    Parser10.pas.915: while LastOP^.NextOperation <> nil do
    00446C4A 837B0C00          cmp dword ptr [ebx+$0c],$00
    00446C4E 75F0              jnz TCustomParser.GetValue + $10
    Parser10.pas.923: LastOP^.MathProc(LastOP);
    00446C50 8BC3              mov eax,ebx
    00446C52 FF5310            call dword ptr [ebx+$10]
    Parser10.pas.924: Result := LastOP^.Dest^;
    00446C55 8B4308            mov eax,[ebx+$08]
    00446C58 DD00              fld qword ptr [eax]
    00446C5A DB3C24            fstp tbyte ptr [esp]
    00446C5D 9B                wait
    00446C5E EB0E              jmp TCustomParser.GetValue + $3E
```

➤ *Figure 1*

➤ *Listing 6*

```
LastOP^.MathProc(LastOP);
LastOP := LastOP^.NextOperation;
LastOP^.MathProc(LastOP);
LastOP := LastOP^.NextOperation;
LastOP^.MathProc(LastOP);
LastOP := LastOP^.NextOperation;
...
LastOP^.MathProc(LastOP);
LastOP := LastOP^.NextOperation;
```

➤ *Listing 7*

```
while LastOP^.NextOperation <> nil do
begin
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
end;
```

➤ *Listing 8*

```
procedure TCustomParser.GenerateDynamicCode(
  OperationCount: integer);
var
  ThisCallOperation : PCallOperation;
  ReturnLastOp      : PReturnLastOp;
  Operation: POperation;
begin
  { Now generate some code dynamically on the heap to call
    the operations }
  if OperationCount > 0 then begin
    { Allocate a memory block of the right size }
    GetMem(DynamicCode,
      (OperationCount * SizeOf(TCallOperation)) +
      SizeOf(TReturnLastOp));
    { Loop through the operations and build code as we go }
    ThisCallOperation := DynamicCode;
    Operation := FStartOperationList;
    while True do begin
      with ThisCallOperation^ do begin
        MOV_EAX     := MovEAXInstruction;
        LastOpAddr  := Operation;
        CALL        := CallInstruction;
        OFFSET      := PChar(@Operation^.MathProc) -
          (PChar(@ThisCallOperation^.CALL) + 5);
      end;
      Inc(ThisCallOperation);
      if Operation^.NextOperation = nil then
        Break;
      Operation := Operation^.NextOperation;
    end;
    { Add code to return the last node }
    ReturnLastOp := PReturnLastOp(ThisCallOperation);
    with ReturnLastOp^ do begin
      MOV_EAX    := MovEAXInstruction;
      LastOpAddr := Operation;
      RET        := RetInstruction;
    end;
  end;
end;
```

This is pretty good code. But, because we will be generating the machine code, we can cut a few corners and reduce the indirection the compiler must make. For instance, we can just plug the address of the routines directly into the CPU instructions instead of loading them indirectly from the operation records at runtime. Again, this will shave off a few CPU cycles.

After some studying and trial-and-error, I was ready to write the machine code generation. Listing 8 shows the `GenerateDynamicCode` method.

After parsing the string expression, and converting it into the linked list of operation records, we have counted the number of operations and this is passed to the routine in the `OperationCount` parameter. The amount of code we must generate is proportional to this number, so we allocate the correct number of bytes using `GetMem` (see the sidebar entitled *Don't Execute Me!*).

In this case, the code we are generating has a pre-defined layout, so it made things easier to define a couple of records that mimic the machine code. I also defined constants for the instruction op-codes. This makes the code more self-documenting. See Listing 9.

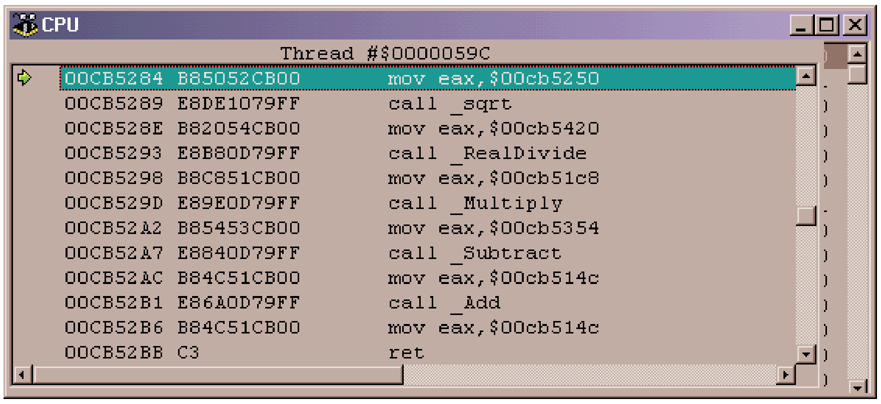
The generated code will consist of a number of `TCallOperation` records (one for each operation) followed by a single `TReturnLastOp` record. The `TCallOperation` records first loads `EAX` with the address of the `Operation` record then calls directly to the procedure implementing the mathematical operation. Each call is unrolled and thus has its own source address. Because of this, the CPU is able to achieve 100% branch target prediction (versus close to 0% for the old code). This is the main reason for the performance improvement. There are also fewer memory accesses, no checks and no branches (except the `CALL`s, of course). The compiled Pascal code had to check for `nil`, load the operation address and also the address of the next operation record. All of this has been unrolled in the generated machine code.

```
while LastOP^.NextOperation <> nil do begin
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
  LastOP^.MathProc(LastOP);
  if LastOP^.NextOperation = nil then Break;
  LastOP := LastOP^.NextOperation;
end;
```

➤ *Listing 9*

Note the special logic we must use to calculate the relative `CALL` instructions. This is because the immediate address value must be a relative offset (negative to `CALL` backwards and positive to `CALL` forwards) to the current `IP` address value. This peculiar looking chip design ensures that the code can be reallocated in memory without requiring fixups. This reduces the number of fixups required when a DLL must be dynamically rebased, for instance. While executing the `CALL`, the `IP` (instruction pointer) is already pointing to the next instruction (5 bytes after this instruction). So we have to take this into account when calculating

➤ *Figure 2*



```
                     Thread #$0000059C
00CB5284 B85052CB00          mov eax,$00cb5250
00CB5289 E8DE1079FF          call _sqrt
00CB528E B82054CB00          mov eax,$00cb5420
00CB5293 E8B80D79FF          call _RealDivide
00CB5298 B8C851CB00          mov eax,$00cb51c8
00CB529D E89E0D79FF          call _Multiply
00CB52A2 B85453CB00          mov eax,$00cb5354
00CB52A7 E8840D79FF          call _Subtract
00CB52AC B84C51CB00          mov eax,$00cb514c
00CB52B1 E86A0D79FF          call _Add
00CB52B6 B84C51CB00          mov eax,$00cb514c
00CB52BB C3                  ret
```

Don't Execute Me!

At this point, avid readers might object and say that we should use the special `TCodeHeap` object from the Delayed DLL Loading[6] article. Recall that we intend to generate machine instructions in this block of memory and the Win32 API documentation says memory pages that will be executed must have the `PAGE_EXECUTE` access protection flag set (see the documentation for `VirtualProtect`, for instance). In my HVDLL project, I followed the recipe, allocated my own private memory pages and set the protection flags to `PAGE_EXECUTE_READWRITE`.

My modified Parser10 code originally used the `TCodeHeap` class and I had written: 'We allocate the correct number of bytes from a specially designed `CodeHeap`. This is a Windows API based heap completely separate from the RTL memory allocator. This is done because we must set the executable attribute on the memory pages we allocate to allow the bytes we patch to be executed by the CPU. Otherwise we would get Access Violations when trying to CALL the dynamically generated code.'

Imagine my surprise when I read this in *Inside Windows 2000*, 3rd Edition, by Mark Russinovich and David Solomon (page 397): 'The x86 architecture doesn't implement execute-only access (that is, code can be executed in any readable page), so Windows 2000 doesn't support this option in any practical sense (though IA-64 does). Windows 2000 treats `PAGE_EXECUTE_READ` as `PAGE_READONLY` and `PAGE_EXECUTE_READWRITE` as `PAGE_READWRITE`.'

All the normal memory allocated with System's `GetMem` has the `PAGE_READ` attribute set, so there is actually no need for the VirtualProtect tricks in my `TCodeHeap` class.

the address, giving us the somewhat complex expression:

```
OFFSET := PChar(
  @OperationLoop^.Operation) -
  (PChar(@ThisCallOperation^.CALL)
  + 5);
```

Figure 2 shows the resulting code in Delphi's CPU view.

In this case the expression was `Sqrt(pi)*4+5-3/5`. The CPU view is able to interpret the addresses and find the symbolic names (`_sqrt`, `_ReadDivide` and so on). Notice how the operations have been re-ordered into the correct evaluation order. This code is very fast and does not suffer from any misprediction stalls at all. We call this lump of code as if it was a function, so after calling all the required `MathProcs`, we set `EAX` to the address of the last operation record and then do a `RET` instruction to return to the caller.

The `DynamicCode` field now points to this block of memory with our dynamically generated 'function' So the evaluation of the expression, corresponding to the loop in Listing 6, can now be written like Listing 10.

```
function TCustomParser.GetValue: Extended;
type
  TCallOperationFunc = function: POperation;
begin
  if Assigned(DynamicCode) then
    Result := TCallOperationFunc(DynamicCode)^.Dest^
  else
    Result := 0;
end;
```

➤ *Above: Listing 10*

➤ *Below: Listing 11*

```
procedure TPlotForm.GoBtnClick(Sender: TObject);
var
  X: Double;
  XTo: Double;
  XStep: Double;
  Y: Double;
begin
  MathExprParser.Expression := ExpressionEdit.Text;
  LineSeries.Clear;
  X     := StrToFloat(XFromEdit.Text);
  XTo   := StrToFloat(XToEdit.Text);
  XStep := StrToFloat(XStepEdit.Text);
  while X <= XTo do begin
    MathExprParser.X := X;
    Y := MathExprParser.Value;
    LineSeries.AddXY(X, Y, '');
    X := X + XStep;
  end;
end;
```

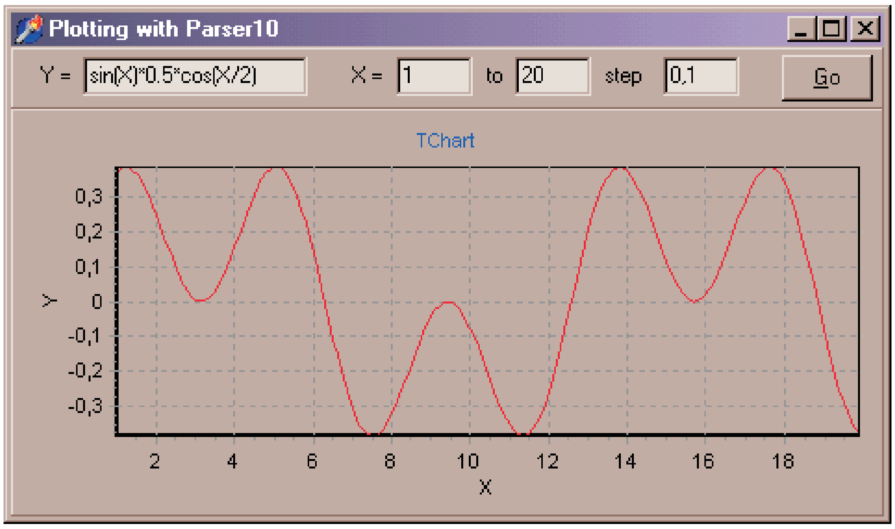
Performance Testing

My claims of improving the performance of Parser10 wouldn't be worth much if I could not back it up with some hard numbers and test applications. The original Parser10 package came with some simple testing projects. A loop that exercises a (somewhat strange) expression showed an improvement from 80ms to 60ms on my machine: that's an improvement of about 25%. I didn't have the time to perform more involved and varied performance testing. To help you carry out your own testing, I have left the old code in place: simply undefine the `DYNAMIC_CODE` define at the top of the `Parser10` unit to get the old looping version of the code.

Plotting XY Functions

A simple and enlightening way of using Parser10 is to plot the X and Y coordinates of various different simple functions. This can be done easily with TeeChart and Parser10, see Figure 3.

➤ *Figure 3*



The meat of the code for this sample can be found in Listing 11. This code isn't exactly bullet-proof, but it's useful as a demonstration. Specifically, if the starting value and step value are too far apart, round-off errors caused by floating point operations will start to show. You can enter any formula that takes X as an input.

This concept can be extended to plot other types of charts or even 3D-surfaces with X, Y and Z axes. Then you would have to include two variables in the formula for Y. Both X and Z would vary from a maximum to a minimum using a specified step value.

Other uses would be to integrate user-defined or database-stored formulae as part of your application. For instance, a financial information package might download the available formulae from a central server in text format. This way the application can be easily extended with new functionality without any recompiles or time-consuming downloads.

Machine-Code Generation As An Optimisation Technique

Most of the time you cannot do much better than the compiler when generating machine code. Most optimisations can be achieved by selecting the right algorithm for the problem at hand

(I'm sure you have heard this a thousand times before, but bear with me). And it goes without saying that you should always measure to find if and where you might have bottlenecks. And measure again to see that your changes actually made an improvement in the performance. 95% of all code will never have to be optimised. However, if you find that your situation fulfils the following criteria, consider if machine-code generation could be the right algorithm to use:

- ➢ You are looping over a structure, calling a virtual method, method pointer or procedure pointer;
- ➢ The routine being called typically varies each time;
- ➢ This loop is called from another external loop, or from time-sensitive code;
- ➢ The structure is relatively static compared to the number of times the code is executed.

These points are the indications of a typical interpreter. The main pitfall is calling different functions from the same spot in a loop. This kills the CPU's branch prediction and performance goes down the drain. You will find that this pattern is often found in highly polymorphic code. In some cases it might help to sort the called object instances by type. This way, the effect is reduced, although not eliminated.

Conclusion

Time is flying and the editorial deadline is looming on the horizon, so I will have to wrap up the article here. Hopefully, I have been able to make a useful contribution to an already speedy expression parser. Remember that this is freeware and you are most welcome to keep on improving and expanding it, sharing the result with the rest of the world. The convoluted and brute-force code in `P10Build` could undoubtedly be improved upon. Dynamic resolution of variable names (using an event) would be useful for some applications. The ability to define user functions at runtime, expanding the available functions dynamically, could be nice. Feel free to join in!

Hallvard Vassbotn is a Senior Systems Developer at Infront AS (visit www.theonlinetrader.com), where he develops systems for distributing real-time financial information over the internet. You can reach Hallvard at hallvard.vassbotn@c2i.net

References

1. Agner Fog: www.agner.org
2. Stefan Hoffmeister: www.econos.de
3. Renate Shcaef: www.xmission.com/~renates/delphi.html
4. Alin Flaidăr: www.datalog.ro/delphi/delphires.html
5. Chris McNeil, *Express Yourself*, The Delphi Magazine Issue 23, July 1997
6. Hallvard Vassbotn, *DelayLoading of DLLs*, Issue 43, March 1999