COM Corner: Efficient Variants

by Steve Teixeira

We haven't been very nice to `Variants` lately. Oh, sure, they were our best friend in Delphi 2, when they were the best and easiest way to call methods on an Automation object. Then along came Delphi 3 with its shiny new interfaces and dispinterfaces, and `Variants` immediately became second class citizens in our COM world. We began to talk about them behind their backs. 'Why on earth would you use a `Variant`, when an interface or dispinterface is so much more efficient?' While `Variant` is the most flexible of the three, it is also the least efficient, because it uses late binding to call Automation methods. Let's review the three types of method call binding that Automation provides.

Early binding. This is the type of binding provided by interfaces. It is generally the most efficient because methods are called directly based on the vtable offset. This means that method offsets must be known at compile-time.

Late binding. This is the type of binding used by Variants. There is no compile-time checking of method calls. Instead, a method name is converted at runtime to a dispid using the `IDispatch.GetIDsOfNames` method, and the resultant dispid is passed to the `IDispatch.Invoke` method to execute it.

ID binding. This is the type of binding used by dispinterfaces. Like late binding, methods are executed through `IDispatch.Invoke` rather directly through the vtable, however, the dispid is known at compile-time, which obviates the need for a runtime call to `GetIDsOfNames`. This makes method calls through ID binding more efficient than late binding, but still less efficient than early binding.

Despite the bad things we may say about `Variants` when they're not around, they can be pretty useful. After all, they enable us to call an Automation method simply by name, without having to link in other files, such as a type library import file. If only they could be made to function more efficiently, perhaps they would find their way back into our good graces.

Like any good developer looking to optimize code, we should first pinpoint exactly what makes the code in question so slow before we try to optimize. Based on the above description of method bindings, it's not difficult to determine that `Variants` are inefficient because they use late binding, which must call `IDispatch.GetIDsOfNames` and `IDispatch.Invoke` for every method call. This is necessary because there needs to be a way to call a method based on a string containing the method name. However, the problem is that `GetIDsOfNames` is called every time you call a function, even if you call the same function 1,000 times! What would be nice is if `GetIDsOfNames` were called only the first time, and `Variant` remembered the dispid for subsequent calls, to shortcut the process, improving performance.

➤ *Table 1*

Field	Purpose
FDispValue	Represents the `IDispatch` pointer value contained within the `Variant`.
FRefCount	Contains the reference count of this class. A reference count is used because it is possible to have multiple `Variants` with references to the same instance of an Automation object.
FDispIds	A string list that contains the name/dispid mappings for the `Variant`. A string list is convenient since it provides a pre-built object that handles the correlation of text and a pointer, and issues such as sorting are handled automatically.
FNext	A pointer to the next object in the linked list.
FPrev	A pointer to the previous object in the linked list.
FOwner	A back-pointer to the owning `TVariantCache` object.

Caching Dispids

You might recall from *COM Corner* in July 1999 (Issue 47) that Delphi provides a means for hooking into the function called by the compiler to invoke Automation methods on `Variants`. The compiler generates code to call Automation methods from `Variants` by calling through the `VarDispProc` pointer in the `System` unit. The `ComObj` unit assigns a procedure called `VarDispInvoke` to this pointer, and it is that procedure that normally handles calls to all Automation methods. If you browse the source code for this method, you will be able to see for yourself that its primary goal in life is to call `GetIDsOfNames` to obtain a dispid from a name and `Invoke` to execute the Automation method. However, since this method is called only through the `VarDispProc` pointer, there is nothing to stop you assigning your own method to this pointer to perform some custom behavior when an Automation method is invoked. In other words, it is a window into a `Variant`'s inner workings that will enable us to cache dispids.

Listing 1 shows the source code for the `VarCache` unit, which contains all the logic necessary for dispid caching (the code is on the disk too). I'll take this unit one step at a time to explain the processes involved in caching dispids and streamlining the `VarDispProc` to take advantage of the cache.

The first step is to create the objects necessary to encapsulate the dispid cache. These are found at the top of the unit in the form of

the TVariantEntry and TVariantCache classes. TVariantEntry encapsulates the properties of a single instance of a Variant. This class contains the private fields shown in Table 1.

This class also contains the GetIdsOfNames method, which gets a name/dispid mapping from the internal string list, returning True on success.

As you may be able to divine from the FNext and FPrev fields, this class is intended to be used in the context of a linked list. The next object in the unit, TVariantCache, encapsulates this list, which is actually a circular doubly-linked list (ie, each node contain pointers to the previous and next nodes in the list, and the first and last nodes point to each other). This structure allows the complete list to be traversed in either direction given a pointer to any node in the list. The FVariantEntries fields of this class contain a reference to a node in the list. For maximum efficiency, the code is written such that this field always contains the node that was most recently added or found. This means multiple calls in a row to the same Variant don't have to walk the list to find the instance: the most recent is always the first.

The TVariantCache class also contains Add, Remove, and Find methods, which work by taking an OleVariant as a parameter and matching it to the correct TVariantEntry in the FVariantEntries list. A global instance, called VariantCache, of the TVariantCache class is created in the initialization code for the VarCache unit.

Once you understand how these two classes work, the CachingVarDispInvoke procedure will make more sense. This is the procedure which is assigned to System's VarDispProc pointer in order to handle Variant Automation calls. It first checks to ensure the Variant contains the correct type. It then looks for the Variant in the VariantCache global list. If found, it attempts to short-circuit the call to IDispatch.GetIDsOfName by looking in its own list for the name/dispid mapping. If found, it directly calls Invoke. Otherwise, it calls IDispatch.GetIDsOfNames and adds the name/dispid pair to the cache.

The remainder of the code in this unit deals with the cleanup of the Variant from the cache when the Variant itself is cleared. This process is quite a bit more complicated than the dispid caching scheme I just discussed. It works by hooking into the System unit's _VarClr procedure and writing instructions over the top of _VarClr in order to cause the flow of execution to jump to my own procedure, called MyVarClear. The compiler automatically generates a call to _VarClr when a Variant is cleared or goes out of scope, so it's a natural place to patch. The code that performs the runtime patch is located in the RemapVarClrProc, which creates a byte array containing the instructions to jump to MyVarClear and writes those instructions over the top of the original contents of _VarClr.

Allow me to be the first to admit that this technique, to put it nicely, is a real kludge, but there is no other way to hook into the clearing

➤ *Listing 1: VarCache.pas*

```
unit VarCache;
interface
uses Windows, Classes, ActiveX;
type
  TVariantCache = class;
  TVariantEntry = class(TObject)
  private
    FDispValue: Pointer;
    FRefCount: Integer;
    FDispIds: TStringList;
    FNext: TVariantEntry;
    FPrev: TVariantEntry;
    FOwner: TVariantCache;
  public
    constructor Create(const V: OleVariant; PrevEntry:
      TVariantEntry; AOwner: TVariantCache);
    destructor Destroy; override;
    function GetIdsOfNames(Names: PChar; DispIDs:
      PDispIDList): Boolean;
  end;
  TVariantCache = class(TObject)
  private
    FVariantEntries: TVariantEntry;
  public
    function Add(const V: OleVariant; Names: PChar; DispIDs:
      PDispIDList; Length: Integer): TVariantEntry;
    procedure Remove(const V: OleVariant);
    function Find(const V: OleVariant): TVariantEntry;
  end;
implementation
uses
  ComObj, SysUtils, ComConst;
type
  TDynDispIDList = array of TDispID;
const
  MemSize = 6;
var
  OldVarClear, NewVarClear: PByte;
  VarClearCode: array[1..MemSize] of Byte;
  VariantCache: TVariantCache;
constructor TVariantEntry.Create(const V: OleVariant;
  PrevEntry: TVariantEntry; AOwner: TVariantCache);
begin
  FOwner := AOwner;
  FDispValue := TVarData(V).VDispatch;
  FRefCount := 1;
  FDispIds := TStringList.Create;
  FDispIds.Duplicates := dupError;
  FDispIds.Sorted := True;
  if PrevEntry <> nil then begin
    FPrev := PrevEntry;
    if PrevEntry.FNext = nil then
      FNext := PrevEntry
    else
      FNext := PrevEntry.FNext;
    FNext.FPrev := Self;
    PrevEntry.FNext := Self;
  end;
end;
destructor TVariantEntry.Destroy;
var
  I: Integer;
  List: TDynDispIdList;
begin
  List := nil;
  for I := FDispIds.Count - 1 downto 0 do begin
    TObject(List) := FDispIds.Objects[I];
    List := nil;
  end;
  FDispIDs.Free;
  inherited Destroy;
  if FNext <> nil then begin
    if FNext.FNext = Self then begin
      FNext.FNext := nil;
      FNext.FPrev := nil;
    end else begin
      FNext.FPrev := FPrev;
      FPrev.FNext := FNext;
    end;
    FOwner.FVariantEntries := FNext;
  end else
    FOwner.FVariantEntries := nil;
end;
function TVariantEntry.GetIdsOfNames(Names: PChar;
  DispIDs: PDispIDList): Boolean;
var
  Index: Integer;
  ArrayObj: TObject;
begin
  Result := FDispIds.Find(Names, Index);
  if Result then begin
    ArrayObj := FDispIds.Objects[Index];
```

{ Continued on page 54...}

{ Continued from page 52}

```
    Move(TDynDispIdList(ArrayObj)[0], DispIDs^,
      Length(TDynDispIdList(ArrayObj)) * SizeOf(TDispID));
  end;
end;
function TVariantCache.Add(const V: OleVariant; Names:
  PChar; DispIDs: PDispIDList; Length: Integer):
  TVariantEntry;
var
  List: TDynDispIdList;
  Index: Integer;
begin
  Result := Find(V);
  SetLength(List, Length);
  Move(DispIDs^, List[0], Length * SizeOf(TDispID));
  if Result <> nil then begin
    Inc(Result.FRefCount);
    if not Result.FDispIds.Find(Names, Index) then
      Result.FDispIds.AddObject(Names, TObject(List));
  end else begin
    Result := TVariantEntry.Create(V,FVariantEntries,Self);
    Result.FDispIds.AddObject(Names, TObject(List));
  end;
  // prevent automatic cleanup of dynamic array
  Pointer(List) := nil;
  FVariantEntries := Result;
end;
function TVariantCache.Find(const V: OleVariant):
  TVariantEntry;
var VarEntry: TVariantEntry;
begin
  Result := nil;
  if FVariantEntries <> nil then begin
    VarEntry := FVariantEntries;
    repeat
      if VarEntry.FDispValue = TVarData(V).VDispatch
        then begin
        Result := VarEntry;
        FVariantEntries := Result;
        Exit;
      end;
      VarEntry := VarEntry.FNext;
    until (VarEntry = FVariantEntries) or (VarEntry = nil);
  end;
end;
procedure TVariantCache.Remove(const V: OleVariant);
var VarEntry: TVariantEntry;
begin
  if TVarData(V).VType = varDispatch then begin
    VarEntry := Find(V);
    if VarEntry <> nil then begin
      Dec(VarEntry.FRefCount);
      if VarEntry.FRefCount = 0 then VarEntry.Free;
    end;
  end;
end;
procedure RemoveVariantFromCache(var V: Variant);
begin
  VariantCache.Remove(V);
end;
procedure MyVarClear;
asm
  push eax                      // save registers
  push edx
  mov edx, eax                  // put Variant in edx
  lea eax, VariantCache         // put VariantCache object
                                // self in eax
  call TVariantCache.Remove     // call Remove
  mov eax, edx                  // put Variant back in eax
  pop edx
  call System.@VarClear // do normal variant clearing logic
  pop eax
end;
procedure RemapVarClrProc;
var
  JmpInst: array[1..MemSize] of Byte;
  OldProtect: DWORD;
  NewPtr: PByte;
begin
  // NewFoo holds addr of MyVarClear
  NewVarClear := @MyVarClear;
  // NewPtr holds addr of NewVarClear
  NewPtr := @NewVarClear;
  // set up array containing opcodes for jump...
  JmpInst[1] := $FF;                        // jmp
  JmpInst[2] := $25;                        // dword ptr
  Move(NewPtr, JmpInst[3], SizeOf(NewPtr)); // [NewFoo]
  // Put address of _VarClr into OldVarClear
  asm
    push eax
    mov eax, offset System.@VarClr
    mov OldVarClear, eax
    pop eax
  end;
  // enable read/write/execute permission on _VarClr code
  Win32Check(VirtualProtect(OldVarClear, MemSize,
    PAGE_EXECUTE_READWRITE, @OldProtect));
  // Read old VarClr code
  Move(OldVarClear^, VarClearCode, MemSize);
  // Patch VarClr with new jmp code
  Move(JmpInst, OldVarClear^, MemSize);
end;
// GetIDsOfNames wrapper taken from ComObj.pas
procedure GetIDsOfNames(const Dispatch: IDispatch; Names:
  PChar; NameCount: Integer; DispIDs: PDispIDList);
  procedure RaiseNameException;
  begin
    raise EOleError.CreateResFmt(@SNoMethod, [Names]);
  end;
type
  PNamesArray = ^TNamesArray;
  TNamesArray = array[0..0] of PWideChar;
var
  N, SrcLen, DestLen: Integer;
  Src: PChar;
  Dest: PWideChar;
  NameRefs: PNamesArray;
  StackTop: Pointer;
  Temp: Integer;
begin
  Src := Names;
  N := 0;
  asm
    MOV  StackTop, ESP
    MOV  EAX, NameCount
    INC  EAX
    SHL  EAX, 2  // sizeof pointer = 4
    SUB  ESP, EAX
    LEA  EAX, NameRefs
    MOV  [EAX], ESP
  end;
  repeat
    SrcLen := StrLen(Src);
    DestLen :=
      MultiByteToWideChar(0, 0, Src, SrcLen, nil, 0) + 1;
    asm
      MOV  EAX, DestLen
      ADD  EAX, EAX
      ADD  EAX, 3      // round up to 4 byte boundary
      AND  EAX, not 3
      SUB  ESP, EAX
      LEA  EAX, Dest
      MOV  [EAX], ESP
    end;
    if N = 0 then
      NameRefs[0] := Dest
     else
      NameRefs[
    NameCount - N] := Dest;
    MultiByteToWideChar(0, 0, Src, SrcLen, Dest, DestLen);
    Dest[DestLen-1] := #0;
    Inc(Src, SrcLen+1);
    Inc(N);
  until N = NameCount;
  Temp := Dispatch.GetIDsOfNames(GUID_NULL, NameRefs,
    NameCount, GetThreadLocale, DispIDs);
  if Temp = Integer(DISP_E_UNKNOWNNAME) then
    RaiseNameException else OleCheck(Temp);
  asm
    MOV  ESP, StackTop
  end;
end;
procedure CachingVarDispInvoke(Result: PVariant; const
  Instance: Variant; CallDesc: PCallDesc; Params: Pointer);
  cdecl;
  procedure RaiseException;
  begin
    raise EOleError.CreateRes(@SVarNotObject);
  end;
var
  Dispatch: Pointer;
  DispIDs: array[0..63] of Integer;
  VarEntry: TVariantEntry;
  Names: PChar;
  Count: Integer;
begin
  if TVarData(Instance).VType = varDispatch then
    Dispatch := TVarData(Instance).VDispatch
  else if TVarData(Instance).VType = (varDispatch or
    varByRef) then
    Dispatch := Pointer(TVarData(Instance).VPointer^)
  else
    RaiseException;
  Names := @CallDesc^.ArgTypes[CallDesc^.ArgCount];
  Count := CallDesc^.NamedArgCount + 1;
  VarEntry := VariantCache.Find(Instance);
  if (VarEntry = nil) or
    (not VarEntry.GetIdsOfNames(Names, @DispIDs)) then begin
    GetIDsOfNames(IDispatch(Dispatch), Names, Count,
      @DispIDs);
    VariantCache.Add(Instance, Names, @DispIDs, Count);
  end;
  if Result <> nil then VarClear(Result^);
  DispatchInvoke(IDispatch(Dispatch), CallDesc, @DispIDs,
    @Params, Result);
end;
initialization
  VariantCache := TVariantCache.Create;
  RemapVarClrProc;
  VarDispProc := @CachingVarDispInvoke;
finalization
  VariantCache.Free;
end.
```

of a Variant in order to clean up the cache. This technique is also highly dependent on how the code is compiled, so the implementation shown in VarCache.pas works only with Delphi 5 (although it would be possible to do this with other versions).

The MyVarClear procedure adjusts the list for the Variant which is being cleared (by decrementing the reference count of, and, if necessary, deleting the TVariantEntry object).

In order to take advantage of the performance improvements provided by dispid caching, I feel it's important that users shouldn't have to change the way they work. For that reason, the VarCache unit is written in such a way that all you need to do is add it to a uses clause in your application in order to receive its benefits. To test the unit, I wrote the few lines of code shown in Listing 2 that communicate with the Word 2000 automation server and perform a single task repeatedly.

Without the VarCache unit in my project, this code takes about 5,500 milliseconds to run on my Pentium II 400MHz machine. However, after adding the VarCache unit to my project, the execution time drops to the 3,200 millisecond neighborhood, showing a hefty efficiency gain for an Automation method which is called repeatedly in a tight loop.

If you have an application which is written such that Automation methods are not called many times, then dispid caching will help little on that particular project. The more individual methods are called on the same Variant, the better performance should be boosted.

➤ *Listing 2*

```
procedure TForm1.Button1Click(
  Sender: TObject);
var
  V1: OleVariant;
  I: Integer;
  T1: Cardinal;
begin
  V1 := CreateOleObject(
    'Word.Application');
  T1 := GetTickCount;
  for I := 1 to 1000 do
    V1.Caption := V1.Caption + 'x';
  Caption :=
    IntToStr(GetTickCount - T1);
  V1.Visible := True;
end;
```

Summary

No longer does Variant have to hang its head as 'that really inefficient way to do Automation.' With the help of dispid caching, Variants can approach the efficiency of dispinterfaces, but with the inherent advantages of truly late binding. Hopefully, this new technique will make it feasible for you to use Variants in cases where you previously thought it prohibitively inefficient. And of course, regardless of how it helps our real work, it's always fun to get under the hood and see how far we can stretch Delphi's Automation capabilities.

Steve Teixeira is the VP of software development at DeVries Data Systems in Silicon Valley, and co-author of the upcoming *Delphi 5 Developer's Guide*. You can reach Steve at steve@dvdata.com