

jul 31, 06 9:17

RTCG_Overhead.cs

Page 1/17

```

/* This program evaluates the overhead of doing runtime code generation (RTCG)
 * in C#. The following methods for are evaluated:
 *
 * - Compiletime compiled (base of comparison)
 * - RTCG expressing the program as a CodeDOM structure
 * - RTCG compiling from source (text) at runtime
 * - RTCG using ILAsm using a dynamicmethod (.NET 2.0)
 * - RTCG using ILAsm using an interface. (.NET 1.x)
 *
 * The following is measured:
 *
 * - Overhead of compiling code
 * - Overhead of calling the compiled code
 *
 * To do so the well known x^y function is implemented in C# and ILAsm
 * and compiled and run according to the methods above. It should be noted that
 * x is a floating point number and y is an integer.
 *
 * Specifically the code is compiled Z times after which the code is compiled on
 * ce
 * but evaluated Z times. As a safeguard the code
 * is finally compiled _and_ evaluated Z times and if
 * the numbers sums up it is concluded that the meassurements were valid.
 *
 * As a last note the above is performed W times and the numbers
 * on which the conclusion is based, is an average of the W runs.
 *
 * NB NB NB: This code uses the _external_ system.managment classes of .NET
 *
 * Copyright Thomas S. Iversen, thomassi@dina.kvl.dk, 2006-03-01, 2006-05-25
 */

```

```

using System;
using System.Collections.Generic;
using Microsoft.CSharp; // CSharpCodeProvider
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Reflection;
using System.Reflection.Emit;
using System.Text;
using System.IO;
using System.Diagnostics; // Stopwatch
using System.Management; // ManagmentClass/Scope/Object
using System.Security.Cryptography; // MD5, hashes in general.

```

```

namespace RTCG_Overhead
{
    public interface IPow
    {
        double pow(double x, int y);
    }

    delegate double PowerDelegate(double x, int y);

    abstract class Method
    {
        private String name;
        private int compilecount;
        private int compilehrcount;
        private int invokecount;
    }
}

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 2/17

```

static public double staticpow(double x, int y)
{
    double r = 1;
    while (y > 0)
    {
        if (y % 2 == 0)
        {
            x = x * x;
            y = y / 2;
        }
        else
        {
            r = r * x;
            y = y - 1;
        }
    }
    return r;
}

public int CompileCount
{
    get { return compilecount; }
}

public int CompileHRCount
{
    get { return compilehrcount; }
}

public int InvokeCount
{
    get { return invokecount; }
}

public String Name
{
    get { return name; }
}

public Method(String name, int compilecount, int invokecount, int compil
ehrcount)
{
    this.name = name;
    this.compilecount = compilecount;
    this.invokecount = invokecount;
    this.compilehrcount = compilehrcount;
}

public abstract void Compile();
public abstract double Pow(double x, int y);
public abstract double CPow(double x, int y);
public abstract void ClearRefList();
public abstract double CompileHoldReferenceAndInvoke(double x, int y);
}

abstract class ILAsmMethod : Method
{
    public ILAsmMethod(String name, int compilecount, int evalcount, int com
pilehrcount) :
        base(name, compilecount, evalcount, compilehrcount) {}
    public void powgen(ILGenerator ilg, int xarg, int yarg)
}

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 3/17

```

{
    // Assume that first local variable is double x
    // Assume that second local variable is int y

    Label elsepart = ilg.DefineLabel();
    Label looptest = ilg.DefineLabel();
    Label loopstart = ilg.DefineLabel();

    // Declare and initialize r to 1 as a local variable
    LocalBuilder r = ilg.DeclareLocal(typeof(double));
    ilg.Emit(OpCodes.Ldc_R8, 1.0);
    ilg.Emit(OpCodes.Stloc, r);

    // Jump to the test of the while loop exit condition
    ilg.Emit(OpCodes.Br_S, looptest);

    // Loopstart
    ilg.MarkLabel(loopstart);
    // y % 2 == 0?
    ilg.Emit(OpCodes.Ldarg, yarg);
    ilg.Emit(OpCodes.Ldc_I4_2);
    ilg.Emit(OpCodes.Rem);
    ilg.Emit(OpCodes.Ldc_I4_0);
    ilg.Emit(OpCodes.Ceq);
    ilg.Emit(OpCodes.Brfalse_S, elsepart);

    // Yes
    // x = x * x
    ilg.Emit(OpCodes.Ldarg, xarg);
    ilg.Emit(OpCodes.Ldarg, xarg);
    ilg.Emit(OpCodes.Mul);
    ilg.Emit(OpCodes.Starg_S, xarg);

    // y = y / 2
    ilg.Emit(OpCodes.Ldarg, yarg);
    ilg.Emit(OpCodes.Ldc_I4_2);
    ilg.Emit(OpCodes.Div);
    ilg.Emit(OpCodes.Starg_S, yarg);
    ilg.Emit(OpCodes.Br_S, looptest);

    ilg.MarkLabel(elsepart);
    // No
    // r = r * x
    ilg.Emit(OpCodes.Ldloc, r);
    ilg.Emit(OpCodes.Ldarg, xarg);
    ilg.Emit(OpCodes.Mul);
    ilg.Emit(OpCodes.Stloc, r);

    // y = y - 1
    ilg.Emit(OpCodes.Ldarg, yarg);
    ilg.Emit(OpCodes.Ldc_I4_1);
    ilg.Emit(OpCodes.Sub);
    ilg.Emit(OpCodes.Starg_S, yarg);

    ilg.MarkLabel(looptest);

    // y > 0?
    ilg.Emit(OpCodes.Ldarg, yarg);
    ilg.Emit(OpCodes.Ldc_I4_0);
    ilg.Emit(OpCodes.Cgt);
    ilg.Emit(OpCodes.Brtrue_S, loopstart);

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 4/17

```

        // return r;
        ilg.Emit(OpCodes.Ldloc, r);
        ilg.Emit(OpCodes.Ret);
    }

    //
    // Method1: static compiletime resolution.
    //
    class CompileTime : Method
    {
        public CompileTime(String name, int compilecount, int evalcount, int compilehrcount)
        :
            base(name, compilecount, evalcount, compilehrcount) {}

        // We do not need to compile at runtime
        public override void Compile() {}
        public override void ClearRefList() {}
        public override double CompileHoldReferenceAndInvoke(double x, int y)
        {
            Compile();
            return Pow(x, y);
        }

        public override double Pow(double x, int y)
        {
            return staticpow(x, y);
        }

        public override double CPow(double x, int y)
        {
            Compile();
            return Pow(x, y);
        }
    }

    //
    // Method2: static compiletime resolution using reflection
    //
    class CompileTimeReflection : Method
    {
        public CompileTimeReflection(String name, int compilecount, int evalcount, int compilehrcount)
        :
            base(name, compilecount, evalcount, compilehrcount) {}

        // We do not need to compile at runtime
        public override void Compile() {}
        public override void ClearRefList() {}
        public override double CompileHoldReferenceAndInvoke(double x, int y)
        {
            Compile();
            return Pow(x, y);
        }

        public override double Pow(double x, int y)
        {
            Type type = typeof(Method);
            MethodInfo method = type.GetMethod("staticpow");
            return (double)method.Invoke(null, new Object[] { x, y });
        }
    }

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 5/17

```

        public override double CPow(double x, int y)
        {
            Compile();
            return Pow(x, y);
        }

//
// Method3: compile the code from a codedom structure
//
class CodeDOMGraph : Method
{
    private CompilerResults cr = null;
    private IPow codedompow = null;
    private List<IPow> list = new List<IPow>();

    public CodeDOMGraph(String name, int compilecount, int evalcount, int compilehrcount)
    :
        base(name, compilecount, evalcount, compilehrcount) {}

    public override void ClearRefList()
    {
        list.Clear();
    }

    public override double CompileHoldReferenceAndInvoke(double x, int y)
    {
        // Compile code
        Compile();
        list.Add(codedompow);
        return codedompow.pow(x, y);
    }

    public override void Compile()
    {
        // Build codegraph

        CodeCompileUnit ccu = new CodeCompileUnit();
        CodeNamespace rtcg_test = new CodeNamespace("RTCG_Overhead");
        rtcg_test.Imports.Add(new CodeNamespaceImport("System"));
        ccu.Namespaces.Add(rtcg_test);
        CodeTypeDeclaration myclass = new CodeTypeDeclaration("myclass");
        myclass.BaseTypes.Add("IPow");
        rtcg_test.Types.Add(myclass);

        CodeConstructor defaultConstructor = new CodeConstructor();
        defaultConstructor.Attributes = MemberAttributes.Public;
        // Adds the constructor to the Members collection of the BaseType.
        myclass.Members.Add(defaultConstructor);

        CodeMemberMethod pow = new CodeMemberMethod();
        pow.Name = "pow";
        pow.Attributes = MemberAttributes.Public;
        pow.ReturnType = new CodeTypeReference(typeof(double));
        pow.Parameters.Add(new CodeParameterDeclarationExpression(typeof(double), "x"));
        pow.Parameters.Add(new CodeParameterDeclarationExpression(typeof(int), "y"));
        pow.Statements.Add(new CodeParameterDeclarationExpression(typeof(double), "r = 1"));

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 6/17

```

        pow.Statements.Add(new CodeIterationStatement(
            // Init expression (notice how to make this empty)
            new CodeExpressionStatement(new CodeSnippetExpression("")),

            // Test for loop
            new CodeBinaryOperatorExpression(
                new CodeVariableReferenceExpression("y"),
                CodeBinaryOperatorType.GreaterThan,
                new CodePrimitiveExpression(0)
            ),

            // Increment expression
            new CodeExpressionStatement(new CodeSnippetExpression("")),

            // Code to execute inside for loop
            new CodeStatement[] {
                // IF (y % 2) == 0
                new CodeConditionStatement(
                    new CodeBinaryOperatorExpression(
                        new CodeBinaryOperatorExpression(
                            new CodeVariableReferenceExpression("y"),
                            CodeBinaryOperatorType.Modulus,
                            new CodePrimitiveExpression(2)
                        ),
                        CodeBinaryOperatorType.ValueEquality,
                        new CodePrimitiveExpression(0)
                    ),
                    // Then
                    new CodeStatement[]
                    {
                        // x = x * x
                        new CodeAssignStatement(
                            new CodeVariableReferenceExpression("x"),
                            new CodeBinaryOperatorExpression(
                                new CodeVariableReferenceExpression("x"),
                                CodeBinaryOperatorType.Multiply,
                                new CodeVariableReferenceExpression("x")
                            )
                        ),
                        // y = y / 2
                        new CodeAssignStatement(
                            new CodeVariableReferenceExpression("y"),
                            new CodeBinaryOperatorExpression(
                                new CodeVariableReferenceExpression("y"),
                                CodeBinaryOperatorType.Divide,
                                new CodePrimitiveExpression(2)
                            )
                        )
                    },
                    // Else
                    new CodeStatement[]
                    {
                        // r = r * x
                        new CodeAssignStatement(
                            new CodeVariableReferenceExpression("r"),
                            new CodeBinaryOperatorExpression(

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 7/17

```

        new CodeVariableReferenceExpression("r"),
        CodeBinaryOperatorType.Multiply,
        new CodeVariableReferenceExpression("x")
    ),
    // y = y - 1

    new CodeAssignStatement(
        new CodeVariableReferenceExpression("y"),
        new CodeBinaryOperatorExpression(
            new CodeVariableReferenceExpression("y"),
            CodeBinaryOperatorType.Subtract,
            new CodePrimitiveExpression(1)
        )
    ),
}
) // CodeIteration Statement
); // Statements.Add
myclass.Members.Add(pow);

pow.Statements.Add(new CodeMethodReturnStatement(new CodeVariableReferenceExpression("r")));

// Code compilation/Source generation
CSharpCodeProvider provider = new CSharpCodeProvider();

// NB: This code can be used to output the CodeDOM graph as
// a sourcecode file:
//
// <START>
//
// IndentedTextWriter tw = new IndentedTextWriter(
// new StreamWriter("c:/sourcecode.cs", false), " ");

// Generate source code using the code provider.
//
// provider.GenerateCodeFromCompileUnit(ccu, tw,
// new CodeGeneratorOptions());

// Close the output file.
//
// tw.Close();
//
// </END>

CompilerParameters cp = new CompilerParameters();
cp.ReferencedAssemblies.Add("system.dll");
cp.ReferencedAssemblies.Add(typeof(IPow).Assembly.Location);
cp.GenerateExecutable = false;
cp.GenerateInMemory = true;

cr = provider.CompileAssemblyFromDom(cp, ccu);

if (cr.Errors.Count > 0)
{
    foreach (CompilerError ce in cr.Errors)
        Console.WriteLine(ce.ToString());
}

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 8/17

```

    }
    else
    {
        Assembly assembly = cr.CompiledAssembly;
        codedompow = (IPow)assembly.CreateInstance("RTCG_Overhead.myclass");
    }
}

public override double Pow(double x, int y)
{
    if (codedompow == null)
        Compile();
    return codedompow.pow(x, y);
}

public override double CPow(double x, int y)
{
    Compile();
    return codedompow.pow(x, y);
}

//
// Method4: Compile from source using the CodeDOM
//
class CodeDOMSrc : Method
{
    public CodeDOMSrc(String name, int compilecount, int evalcount, int compilehrcount)
    :
        base(name, compilecount, evalcount, compilehrcount) {}

    private CompilerResults cr = null;
    private IPow codedompow = null;
    private List<IPow> list = new List<IPow>();

    public override void ClearRefList()
    {
        list.Clear();
    }

    public override double CompileHoldReferenceAndInvoke(double x, int y)
    {
        // Compile code
        Compile();
        list.Add(codedompow);
        return codedompow.pow(x, y);
    }

    public override void Compile()
    {
        string src = "using System;" +
            "class myclass:RTCG_Overhead.IPow" +
            "{" +
            "public myclass(){}" +
            "public double pow(double x, int y)" +
            "{" +
            "double r = 1;" +
            "while (y > 0)" +
            "{" +
            "if (y % 2 == 0)" +

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 9/17

```

        "{" +
        "x = x * x;" +
        "y = y / 2;" +
        "}" +
        "else" +
        "{" +
        "r = r * x;" +
        "y = y - 1;" +
        "}" +
        "}" +
        "return r;" +
        "}" +
        "};";

CSharpCodeProvider provider = new CSharpCodeProvider();
CompilerParameters cp = new CompilerParameters();
cp.ReferencedAssemblies.Add("system.dll");
cp.ReferencedAssemblies.Add(typeof(IPow).Assembly.Location);
cp.GenerateExecutable = false;
cp.GenerateInMemory = true;

cr = provider.CompileAssemblyFromSource(cp, src);

if (cr.Errors.Count > 0)
{
    foreach (CompilerError ce in cr.Errors)
        Console.WriteLine(ce.ToString());
}
else
{
    Assembly assembly = cr.CompiledAssembly;
    codedompow = (IPow)assembly.CreateInstance("myclass");
}

public override double Pow(double x, int y)
{
    if (codedompow == null)
        Compile();
    return codedompow.pow(x, y);
}

public override double CPow(double x, int y)
{
    Compile();
    return codedompow.pow(x, y);
}

//
// Method5: Dynamic method invoke
//
sealed class DynamicMethodInvoke : ILAsmMethod
{
    private DynamicMethod dm = null;
    private List<DynamicMethod> list = new List<DynamicMethod>();
    public DynamicMethodInvoke(String name, int compilecount, int evalcount,
int compilehrcount)
    :
        base(name, compilecount, evalcount, compilehrcount) {}
}

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 10/17

```

public override void ClearRefList()
{
    list.Clear();
}

public override double CompileHoldReferenceAndInvoke(double x, int y)
{
    Compile();
    list.Add(dm);
    return (double)dm.Invoke(new Object(), BindingFlags.Default, null, n
ew Object[] { x, y }, null);
}

public override void Compile()
{
    dm = new DynamicMethod("pow", typeof(double), new Type[] { typeof(do
uble), typeof(int) }, typeof(String).Module);
    ILGenerator ilg = dm.GetILGenerator();
    powgen(ilg, 0, 1);
}

public override double CPow(double x, int y)
{
    Compile();
    return (double)dm.Invoke(new Object(), BindingFlags.Default, null, n
ew Object[] { x, y }, null);
}

public override double Pow(double x, int y)
{
    if (dm == null)
        Compile();
    return (double)dm.Invoke(new Object(), BindingFlags.Default, null, n
ew Object[] { x, y }, null);
}

//
// Method6: Dynamic method delegate
//

class DynamicMethodDelegate : ILAsmMethod
{
    private PowerDelegate powdel = null;
    private DynamicMethod dm = null;
    private List<DynamicMethod> list = new List<DynamicMethod>();
    public DynamicMethodDelegate(String name, int compilecount, int evalcoun
t, int compilehrcount)
    :
        base(name, compilecount, evalcount, compilehrcount) {}

    public override void Compile()
    {
        dm = new DynamicMethod("pow", typeof(double), new Type[] { typeof(do
uble), typeof(int) }, typeof(String).Module);
        ILGenerator ilg = dm.GetILGenerator();
        powgen(ilg, 0, 1);
        powdel = (PowerDelegate)dm.CreateDelegate(typeof(PowerDelegate));
    }

    public override void ClearRefList()
    {
}

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 11/17

```

        list.Clear();
    }

    public override double CompileHoldReferenceAndInvoke(double x, int y)
    {
        Compile();
        list.Add(dm);
        return powdel(x, y);
    }

    public override double CPow(double x, int y)
    {
        Compile();
        return powdel(x, y);
    }

    public override double Pow(double x, int y)
    {
        if (dm == null)
            Compile();
        return powdel(x, y);
    }
}

//
// Method7: Dynamic method interface
//
class MethodInterface : ILAsmMethod
{
    private ModuleBuilder moduleBuilder = null;
    private long counter = 0;
    private Type ty = null;
    private IPow IObj = null;
    private List<IPow> list = new List<IPow>();
    private String classname = null;

    private String md5sum(String str)
    {
        byte[] input = Encoding.UTF8.GetBytes(str);
        byte[] output = MD5.Create().ComputeHash(input);
        return Convert.ToBase64String(output);
    }

    public override void ClearRefList()
    {
        list.Clear();
    }

    public override double CompileHoldReferenceAndInvoke(double x, int y)
    {
        // Compile code
        Compile();
        Object obj = ty.GetConstructor(new Type[] { }).Invoke(new Object[] {
});

        IPow IObj = (IPow)obj;
        list.Add(IObj);

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 12/17

```

        return IObj.pow(x, y);
    }

    public MethodInterface(String name, int compilecount, int evalcount, int
compilehrcount)
    :
        base(name, compilecount, evalcount, compilehrcount) {}

    public override void Compile()
    {
        if (moduleBuilder == null)
        {
            // This is the first time we try to create an assembly
            // with a module.

            AssemblyName assemblyName = new AssemblyName();
            assemblyName.Name = "myassembly";
            // Build: run-only assembly
            AssemblyBuilder assemblyBuilder =
                AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
                    AssemblyBuilderA
ccess.Run);

            // Build: module mymodule
            moduleBuilder =
                assemblyBuilder.DefineDynamicModule("mymodule");
        }
        // Build: public class MyClass : IPow { ... }

        classname = "MyClass" + counter++;

        TypeBuilder typeBuilder =
            moduleBuilder.DefineType(classname,
                TypeAttributes.Class | TypeAttributes.Pub
lic,
                typeof(Object));
        typeBuilder.AddInterfaceImplementation(typeof(IPow));
        { // Build constructor: public MyClass() : base() { }
            ConstructorBuilder constructorBuilder =
                typeBuilder.DefineConstructor(MethodAttributes.Public,
                    CallingConventions.HasThis,
                    new Type[] { });
            ILGenerator ilg = constructorBuilder.GetILGenerator();
            ilg.Emit(OpCodes.Ldarg_0); // push this
            ilg.Emit(OpCodes.Call, typeof(Object).GetConstructor(new Type[]
{ }));
        }
        ilg.Emit(OpCodes.Ret);

        // Build method: public virtual double pow(double x, int y) { ..
        MethodBuilder methodBuilder =
            typeBuilder.DefineMethod("pow",
                MethodAttributes.Virtual | MethodAttr
ibutes.Public,
                typeof(double),
                new Type[] { typeof(double), typeof(i
nt) });

        // Obtain an IL generator to build the method body

```

jul 31, 06 9:17

RTCG_Overhead.cs

Page 13/17

```

        ILGenerator ilg = methodBuilder.GetILGenerator();
        powgen(ilg, 1, 2);
    }
    ty = typeBuilder.CreateType();
    Object obj = ty.GetConstructor(new Type[] { }).Invoke(new Object[] {
});

    IObj = (IPow)obj;
}

public override double CPow(double x, int y)
{
    Compile();
    return IObj.pow(x, y);
}

public override double Pow(double x, int y)
{
    if (ty == null)
        Compile();
    return IObj.pow(x, y);
}

}

class Program
{
    static double x = 3;
    static int y = 11;

    static int z = 1000000;
    static int z10 = z * 10;
    static int z100 = z * 100;
    static int slowerz = z / 10;
    static int slowz = z / 1000;
    static int veryslowz = z / 100000;
    static int w = 5;

    static void CompileBenchmark(List<Method> methods)
    {
        Stopwatch watch = new Stopwatch();

        long totalms;
        foreach (Method method in methods)
        {
            totalms = 0;
            Console.WriteLine("Doing {0} compiles ({1})", method.CompileCount, method.Name);
            for (int i = 0; i < w; i++)
            {
                watch.Reset();
                watch.Start();
                for (int j = method.CompileCount; j > 0; j--)
                {
                    method.Compile();
                }
                watch.Stop();
                totalms += watch.ElapsedMilliseconds;

                Console.WriteLine("Run {0}.: {1} ms", i, watch.ElapsedMilliseconds.ToString());
            }
        }
    }
}

```

mandag juli 31, 2006

../TinyCalc/TinyCalc/OverheadBenchmark/Scripts/RTCG_Overhead.cs

jul 31, 06 9:17

RTCG_Overhead.cs

Page 14/17

```

        }
        Console.WriteLine("-----");
        Console.WriteLine("Total: {0}ms for {1} compiles", totalms, w * method.CompileCount);
        Console.WriteLine("Average: {0}ms pr. compile", ((double)totalms) / (w * method.CompileCount));
        Console.WriteLine();
    }
}

static void InvokeBenchmark(List<Method> methods)
{
    Stopwatch watch = new Stopwatch();

    long totalms;
    double r = 0;
    foreach (Method method in methods)
    {
        totalms = 0;
        Console.WriteLine("Doing {0} invocations ({1})", method.InvokeCount, method.Name);
        for (int i = 0; i < w; i++)
        {
            watch.Reset();
            watch.Start();
            for (int j = method.InvokeCount; j > 0; j--)
            {
                r = method.Pow(x, y);
            }
            watch.Stop();
            totalms += watch.ElapsedMilliseconds;

            Console.WriteLine("Run {0}.: {1} ms", i, watch.ElapsedMilliseconds.ToString());
        }
        Console.WriteLine("r = {0}", r);
        Console.WriteLine("-----");
        Console.WriteLine("Total: {0}ms for {1} invocations", totalms, w * method.InvokeCount);
        Console.WriteLine("Average: {0}ms pr. invocations", ((double)totalms) / (w * method.InvokeCount));
        Console.WriteLine();
    }
}

static void CompileHoldReferenceAndInvokeBenchmark(List<Method> methods, PerformanceCounter PCMemoryAvailable, PerformanceCounter PCCommittedBytes)
{
    Stopwatch watch = new Stopwatch();
    double r=0;
    long totalms;
    int w1 = w * 4;
    foreach (Method method in methods)
    {
        totalms = 0;

        method.ClearRefList();
        Console.WriteLine("Doing {0} times {1} (compile, evaluation) pair");
    }
}

```

7/22

```

jul 31, 06 9:17      RTCG_Overhead.cs      Page 15/17
rs (holding a reference) ({2})", w1, method.CompileHRCCount, method.Name);
    Console.WriteLine("Memory available at start: {0}MB", PCMemoryAv
ailable.NextValue());
    Console.WriteLine("Committed bytes at start: {0}", Math.Floor(PC
CommittedBytes.NextValue()));
    Console.WriteLine();
    for (int i = 0; i < w1; i++)
    {
        watch.Reset();
        watch.Start();
        for (int j = method.CompileHRCCount; j > 0 ; j--)
        {
            r = method.CompileHoldReferenceAndInvoke(x, y);
        }
        watch.Stop();
        totalms += watch.ElapsedMilliseconds;

        Console.WriteLine("Run {0}.: {1} ms (Availiable Memory: {2}
MB, Committed Memory: {3} bytes)", i, watch.ElapsedMilliseconds.ToString(), PCMe
memoryAvailable.NextValue(), Math.Floor(PCCommittedBytes.NextValue()));
    }
    Console.WriteLine("r = {0}", r);
    Console.WriteLine("-----");
    Console.WriteLine("Total: {0}ms for {1} compile and invocations
(holding references)", totalms, w1 * method.CompileHRCCount);
    Console.WriteLine("Average: {0}ms pr. compile and invocations (h
olding references)", ((double)totalms) / (w1 * method.CompileHRCCount));

    method.ClearRefList();
    Console.WriteLine();
}
}

static List<PropertyDataCollection> WMIQuery(string queryObject)
{
    List<PropertyDataCollection> list = new List<PropertyDataCollection>
();
    ManagementObjectSearcher searcher;
    try
    {
        searcher = new ManagementObjectSearcher("SELECT * FROM " + query
Object);
        foreach (ManagementObject wmi_HD in searcher.Get())
        {
            list.Add(wmi_HD.Properties);
        }
    }
    catch
    {
        return null;
    }
    return list;
}

static long PhysicalMemoryInstalled()
{
    List<PropertyDataCollection> pdcl = WMIQuery("Win32_PhysicalMemory")
;
    long total = 0;
    if (pdcl != null)
    {

```

```

jul 31, 06 9:17      RTCG_Overhead.cs      Page 16/17
        foreach (PropertyDataCollection pdc in pdcl)
        {
            foreach (PropertyData pd in pdc)
            {
                if (pd.Name.Equals("Capacity"))
                    total += long.Parse(pd.Value.ToString());
            }
        }
    }
    return total / (1024 * 1024);
}

static String ProcessorInformation()
{
    List<PropertyDataCollection> pdcl = WMIQuery("Win32_Processor");

    List<String> interesstingProperties = new List<string>();

    interesstingProperties.Add("Name");
    interesstingProperties.Add("CurrentClockSpeed");
    interesstingProperties.Add("MaxClockSpeed");
    //interesstingProperties.Add("L2CacheSize");
    //interesstingProperties.Add("L2CacheSpeed");

    StringBuilder sb = new StringBuilder();
    if (pdcl != null)
    {
        foreach (String property in interesstingProperties)
        {
            foreach (PropertyDataCollection pdc in pdcl)
            {
                foreach (PropertyData pd in pdc)
                {
                    if (pd.Name.Equals(property))
                    {
                        sb.Append("  ");
                        sb.Append(pd.Name);
                        sb.Append(": ");
                        sb.Append(pd.Value.ToString().Trim());
                        sb.AppendLine();
                    }
                }
            }
        }
    }
    return sb.ToString();
}

static void Main(string[] args)
{
    if (args.Length == 1)
    {
        if (args[0].Equals("Compile") ||
            args[0].Equals("Invoke") ||
            args[0].Equals("CompileRef"))
        {
            PerformanceCounter PCMemoryAvailable = new PerformanceCounte
r("Memory", "Available Mbytes");
            PerformanceCounter PCCommittedBytes = new PerformanceCounter
("Memory", "Committed Bytes");

```


jul 31, 06 9:17

SimpleStaticMathFunction.cs

Page 1/2

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Simple Math function, static argument (SIN(PI()/4))";
            String author = "Thomas S. Iversen";

            // Number of Instances: 8192 * 32 = 262144
            int cols = 32; int rows = 8192;
            int startcol = 1;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=SIN(PI()/4)");

            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols+1, rows);

            sheet.AddCell(sb.ToString(), startcol, 0);
            Cell cell = sheet.CopyCell(startcol, 0);

            for (int col = startcol; col < cols+startcol; col++)
            {
                for (int row = 0; row < rows; row++)
                {
                    if (!(col == startcol && row == 0))
                        sheet.PasteCell(cell, col, row);
                }
            }

            Benchmark benchmark = new Benchmark(title, author);

            XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
            GeneratorOptions.UseFormulaSharing = true;

```

jul 31, 06 9:17

SimpleStaticMathFunction.cs

Page 2/2

```

        for (GeneratorLevel level = GeneratorLevel.Level0; level <= GeneratorLevel.Level7; level++)
        {
            // Skip Level1
            if (level == GeneratorLevel.Level1)
                continue;

            GeneratorOptions.Level = level;

            XmlElement data = benchmark.AddData(dataset, level.ToString());
            XmlElement subdata = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
            for (int run = 1; run <= runs; run++)
            {
                watch.Reset();
                watch.Start();
                wb.Recompute();
                watch.Stop();

                benchmark.AddRun(subdata, run, watch.ElapsedMilliseconds.ToString());
            }

            Console.WriteLine("Value: {0}", sheet.ShowValue(startcol, 0));
        }

        if (filename != null)
        {
            benchmark.Save(filename);
        }
    }
}

```

jul 31, 06 9:17

LongReferenceChainsDoubleSum.cs

Page 1/2

```
// Generate a long series of references
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output)");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "LongReferenceChainsFastsum";
            String author = "Thomas S. Iversen";

            // Number of Instances: 12288 * 2 = 24576
            int rows = 12288; int cols = 2;
            int startcol = 0;
            int runs = 3;
            GeneratorLevel startlevel = GeneratorLevel.Level0;
            GeneratorLevel endlevel = GeneratorLevel.Level3;

            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols + startcol, rows);

            // A1;
            sheet.AddCell("0.5", 0, 0);

            // A2
            sheet.AddCell("=A1*1.00001", 0, 1);

            // A3..A<rows>
            Cell cellA2 = sheet.CopyCell(0, 1);

            for (int row = 2; row < rows; row++)
            {
                sheet.PasteCell(cellA2, 0, row);
            }

            // B1;
            sheet.AddCell("=DOUBLESUM(A$1:A1)", 1, 0);
```

jul 31, 06 9:17

LongReferenceChainsDoubleSum.cs

Page 2/2

```
// B2 .. B<rows>
Cell cellB1 = sheet.CopyCell(1, 0);
for (int row = 1; row < rows; row++)
{
    sheet.PasteCell(cellB1, 1, row);
}

Benchmark benchmark = new Benchmark(title, author);

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = startlevel; level <= endlevel; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, "TinyCalc DOUBLESUM
-- " + level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
    }
    Console.WriteLine("Value: {0}", sheet.ShowValue(1, 0));
    Console.WriteLine("Value: {0}", sheet.ShowValue(0, 1));
    Console.WriteLine("Value: {0}", sheet.ShowValue(0, rows-1));
    Console.WriteLine("Value: {0}", sheet.ShowValue(1, rows-1));
}

if (filename != null)
{
    benchmark.Save(filename);
}
}
```

jul 31, 06 9:17

SimpleReferenceMathFunction.cs

Page 1/2

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Simple Math function, reference argument (SIN($A$1))";
            String author = "Thomas S. Iversen";

            // Number of Instances: 8192 * 32 = 262144
            int cols = 32; int rows = 8192;
            int startcol = 1;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=SIN($A$1)");

            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols+1, rows);

            sheet.AddCell(sb.ToString(), startcol, 0);
            Cell cell = sheet.CopyCell(startcol, 0);

            sheet.AddCell("=PI()/4", 0, 0);

            for (int col = startcol; col < cols+startcol; col++)
            {
                for (int row = 0; row < rows; row++)
                {
                    if (!(col == startcol && row == 0))
                        sheet.PasteCell(cell, col, row);
                }
            }

            Benchmark benchmark = new Benchmark(title, author);

```

jul 31, 06 9:17

SimpleReferenceMathFunction.cs

Page 2/2

```

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = GeneratorLevel.Level0; level <= GeneratorLevel.Level7; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
        Console.WriteLine("Value: {0}", sheet.ShowValue(startcol, 0));
    }

    if (filename != null)
    {
        benchmark.Save(filename);
    }
}

```

jul 31, 06 9:17

TaylorAllReferencesOptimized.cs

Page 1/3

```
// Generate formula for taylor approximation in y terms
// E^x, where x=0.5. y depends on whether the sheet is to be
// calculated in tincalc or excel:

// y = 60 is max for the codegenerator. What is the maxsize bitwise for the code
// generation?
// y = 13 is max for what excel can handle in a single formula (1024 chars)

// Expected value: exp(0.5) = 1.64872127

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for
the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Taylorexansion of exp(A1), Both enumerator (A1) and
\ndenominator (factorial) are referenced. Optimized";
            String author = "Thomas S. Iversen";

            // Number of Instances: 4096 * 32 = 131072
            int cols = 32; int rows = 4096;
            int startcol = 2;

            string x = "$A$1";
            int y = 13;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=1");
            for (int j = 1; j <= y; j++)
            {
                sb.Append("+");
                sb.Append("$A$" + j.ToString());
                sb.Append("/");
                sb.Append("$B$" + j.ToString());
            }

```

jul 31, 06 9:17

TaylorAllReferencesOptimized.cs

Page 2/3

```
//return;

Workbook wb = new Workbook();
Sheet sheet = new Sheet(wb, cols + startcol, rows);

// A1;
sheet.AddCell("0.5", 0, 0);

// A2-A13
for(int i = 2 ; i<= 13; i++)
    sheet.AddCell("=$A$1^" + i.ToString(), 0, i-1);

// B1-B13

sheet.AddCell("1.0", 1, 0);
sheet.AddCell("=B1*2", 1, 1);
sheet.AddCell("=B2*3", 1, 2);
sheet.AddCell("=B3*4", 1, 3);
sheet.AddCell("=B4*5", 1, 4);
sheet.AddCell("=B5*6", 1, 5);
sheet.AddCell("=B6*7", 1, 6);
sheet.AddCell("=B7*8", 1, 7);
sheet.AddCell("=B8*9", 1, 8);
sheet.AddCell("=B9*10", 1, 9);
sheet.AddCell("=B10*11", 1, 10);
sheet.AddCell("=B11*12", 1, 11);
sheet.AddCell("=B12*13", 1, 12);

sheet.AddCell(sb.ToString(), startcol, 0);
Cell cell = sheet.CopyCell(startcol, 0);

for (int col = startcol; col < cols+startcol; col++)
{
    for (int row = 0; row < rows; row++)
    {
        if (!(col == startcol && row == 0))
            sheet.PasteCell(cell, col, row);
    }
}

Benchmark benchmark = new Benchmark(title, author);

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = GeneratorLevel.Level0; level <= Generato
rLevel.Level7; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation tim
e", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();

```

jul 31, 06 9:17

TaylorAllReferencesOptimized.cs

Page 3/3

```
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.To
String());
    }
    Console.WriteLine("Value: {0}", sheet.ShowValue(startcol, 0));
}
if (filename != null)
{
    benchmark.Save(filename);
}
}
```

jul 31, 06 9:17

TaylorReferenceArgument.cs

Page 1/3

```
// Generate formula for taylor approximation in y terms
// E^x, where x=0.5. y depends on whether the sheet is to be
// calculated in tincalc or excel:

// y = 60 is max for the codegenerator. What is the maxsize bitwise for the codegeneration?
// y = 13 is max for what excel can handle in a single formula (1024 chars)

// Expected value: exp(0.5) = 1.64872127

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output)");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Taylorexansion of exp(A1), A1 is referenced";
            String author = "Thomas S. Iversen";

            // Number of Instances: 4096 * 32 = 131072
            int cols = 32; int rows = 4096;

            string x = "$A$1";
            int y = 13;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=1");
            for (int j = 1; j <= y; j++)
            {
                sb.Append("+");
                sb.Append("(");
                for (int i = 1; i <= j; i++)
                {
                    sb.Append(x);
                    if (i < j)
                        sb.Append("*");
                }
            }
        }
    }
}
```

jul 31, 06 9:17

TaylorReferenceArgument.cs

Page 2/3

```
        sb.Append(")");
        for (int i = 1; i <= j; i++)
        {
            sb.Append(i);
            if (i < j)
                sb.Append("*");
        }
        sb.Append(")");
    }

    Workbook wb = new Workbook();
    Sheet sheet = new Sheet(wb, cols+1, rows);
    sheet.AddCell("0.5", 0, 0);
    sheet.AddCell(sb.ToString(), 1, 0);
    Cell cell = sheet.CopyCell(1, 0);

    for (int col = 1; col < cols+1; col++)
    {
        for (int row = 0; row < rows; row++)
        {
            if (!(col == 1 && row == 0))
                sheet.PasteCell(cell, col, row);
        }
    }

    Benchmark benchmark = new Benchmark(title, author);

    XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
    GeneratorOptions.UseFormulaSharing = true;
    for (GeneratorLevel level = GeneratorLevel.Level0; level <= GeneratorLevel.Level7; level++)
    {
        // Skip Level1
        if (level == GeneratorLevel.Level1)
            continue;

        GeneratorOptions.Level = level;

        XmlElement data = benchmark.AddData(dataset, level.ToString());
        XmlElement subdata = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
        for (int run = 1; run <= runs; run++)
        {
            watch.Reset();
            watch.Start();
            wb.Recompute();
            watch.Stop();

            benchmark.AddRun(subdata, run, watch.ElapsedMilliseconds.ToString());
        }
        Console.WriteLine("Value: {0}", sheet.ShowValue(1, 0));
    }
    if (filename != null)
    {
        benchmark.Save(filename);
    }
}
```

```
}
```

jul 31, 06 9:17

LongReferenceChains.cs

Page 1/2

```
// Generate a long series of references

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "LongReferenceChains";
            String author = "Thomas S. Iversen";

            // Number of Instances: 12288 * 2 = 24576
            int rows = 12288; int cols = 2;
            int startcol = 0;
            int runs = 3;
            GeneratorLevel startlevel = GeneratorLevel.Level0;
            GeneratorLevel endlevel = GeneratorLevel.Level7;

            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols + startcol, rows);

            // A1;
            sheet.AddCell("0.5", 0, 0);

            // A2
            sheet.AddCell("=A1*1.00001", 0, 1);

            // A3..A<rows>
            Cell cellA2 = sheet.CopyCell(0, 1);

            for (int row = 2; row < rows; row++)
            {
                sheet.PasteCell(cellA2, 0, row);
            }

            // B1;
            sheet.AddCell("=SUM(A$1:A1)", 1, 0);
```

jul 31, 06 9:17

LongReferenceChains.cs

Page 2/2

```
// B2 .. B<rows>
Cell cellB1 = sheet.CopyCell(1, 0);
for (int row = 1; row < rows; row++)
{
    sheet.PasteCell(cellB1, 1, row);
}

Benchmark benchmark = new Benchmark(title, author);

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = startlevel; level <= endlevel; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, "TinyCalc SUM -- "
+ level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {

        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());

        Console.WriteLine("Value: {0}", sheet.ShowValue(1, 0));
        Console.WriteLine("Value: {0}", sheet.ShowValue(0, 1));
        Console.WriteLine("Value: {0}", sheet.ShowValue(0, rows-1));
        Console.WriteLine("Value: {0}", sheet.ShowValue(1, rows-1));
    }

    if (filename != null)
    {
        benchmark.Save(filename);
    }
}
}
```

jul 31, 06 9:17

TaylorAllReferences.cs

Page 1/3

```
// Generate formula for taylor approximation in y terms
// E^x, where x=0.5. y depends on whether the sheet is to be
// calculated in tincalc or excel:

// y = 60 is max for the codegenerator. What is the maxsize bitwise for the code
// generation?
// y = 13 is max for what excel can handle in a single formula (1024 chars)

// Expected value: exp(0.5) = 1.64872127

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for
the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Taylorexansion of exp(A1), Both enumerator (A1) and
denominator (factorial) are referenced";
            String author = "Thomas S. Iversen";

            // Number of Instances: 4096 * 32 = 131072
            int cols = 32; int rows = 4096;
            int startcol = 2;

            string x = "$A$1";
            int y = 13;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=1");
            for (int j = 1; j <= y; j++)
            {
                sb.Append("+");
                sb.Append("(");
                for (int i = 1; i <= j; i++)
                {
                    sb.Append(x);
                    if (i < j)

```

jul 31, 06 9:17

TaylorAllReferences.cs

Page 2/3

```
                sb.Append(")");
            }
            sb.Append(")");
            for (int i = 1; i <= j; i++)
            {
                sb.Append("$B$" + i.ToString());
                if (i < j)
                    sb.Append(")");
            }
            sb.Append(")");
        }

        Workbook wb = new Workbook();
        Sheet sheet = new Sheet(wb, cols + startcol, rows);

        // A1;
        sheet.AddCell("0.5", 0, 0);

        // B1-B13

        sheet.AddCell("1.0", 1, 0);
        sheet.AddCell("2.0", 1, 1);
        sheet.AddCell("3.0", 1, 2);
        sheet.AddCell("4.0", 1, 3);
        sheet.AddCell("5.0", 1, 4);
        sheet.AddCell("6.0", 1, 5);
        sheet.AddCell("7.0", 1, 6);
        sheet.AddCell("8.0", 1, 7);
        sheet.AddCell("9.0", 1, 8);
        sheet.AddCell("10.0", 1, 9);
        sheet.AddCell("11.0", 1, 10);
        sheet.AddCell("12.0", 1, 11);
        sheet.AddCell("13.0", 1, 12);

        sheet.AddCell(sb.ToString(), startcol, 0);
        Cell cell = sheet.CopyCell(startcol, 0);

        for (int col = startcol; col < cols+startcol; col++)
        {
            for (int row = 0; row < rows; row++)
            {
                if (!(col == startcol && row == 0))
                    sheet.PasteCell(cell, col, row);
            }
        }

        Benchmark benchmark = new Benchmark(title, author);

        XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
        GeneratorOptions.UseFormulaSharing = true;
        for (GeneratorLevel level = GeneratorLevel.Level0; level <= Generator
Level.Level7; level++)
        {
            // Skip Level1
            if (level == GeneratorLevel.Level1)
                continue;

            GeneratorOptions.Level = level;

```

jul 31, 06 9:17

TaylorAllReferences.cs

Page 3/3

```
XmlElement data = benchmark.AddData(dataset, level.ToString());
XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
    }
    Console.WriteLine("Value: {0}", sheet.ShowValue(startcol, 0));
}

if (filename != null)
{
    benchmark.Save(filename);
}
}
```

jul 31, 06 9:17

LongReferenceChainsFastsum.cs

Page 1/2

```
// Generate a long series of references
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output)");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "LongReferenceChainsFastsum";
            String author = "Thomas S. Iversen";

            // Number of Instances: 12288 * 2 = 24576
            int rows = 12288; int cols = 2;
            int startcol = 0;
            int runs = 3;
            GeneratorLevel startlevel = GeneratorLevel.Level0;
            GeneratorLevel endlevel = GeneratorLevel.Level5;

            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols + startcol, rows);

            // A1;
            sheet.AddCell("0.5", 0, 0);

            // A2
            sheet.AddCell("=A1*1.00001", 0, 1);

            // A3..A<rows>
            Cell cellA2 = sheet.CopyCell(0, 1);

            for (int row = 2; row < rows; row++)
            {
                sheet.PasteCell(cellA2, 0, row);
            }

            // B1;
            sheet.AddCell("=FASTSUM(A$1:A1)", 1, 0);

```

jul 31, 06 9:17

LongReferenceChainsFastsum.cs

Page 2/2

```
// B2 .. B<rows>
Cell cellB1 = sheet.CopyCell(1, 0);
for (int row = 1; row < rows; row++)
{
    sheet.PasteCell(cellB1, 1, row);
}

Benchmark benchmark = new Benchmark(title, author);

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = startlevel; level <= endlevel; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, "TinyCalc FASTSUM - " + level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
    }
    Console.WriteLine("Value: {0}", sheet.ShowValue(1, 0));
    Console.WriteLine("Value: {0}", sheet.ShowValue(0, 1));
    Console.WriteLine("Value: {0}", sheet.ShowValue(0, rows-1));
    Console.WriteLine("Value: {0}", sheet.ShowValue(1, rows-1));
}

if (filename != null)
{
    benchmark.Save(filename);
}
}

```

jul 31, 06 9:17

TaylorNoReferences.cs

Page 1/3

```
// Generate formula for taylor approximation in y terms
// E^x, where x=0.5. y depends on whether the sheet is to be
// calculated in tincalc or excel:

// y = 60 is max for the codegenerator. What is the maxsize bitwise for the code
// generation?
// y = 13 is max for what excel can handle in a single formula (1024 chars)

// Expected value: exp(0.5) = 1.64872127

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch

namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for
the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "Taylorexansion of exp(0.5), no cellreferences";
            String author = "Thomas S. Iversen";

            // Number of Instances: 4096 * 32 = 131072
            int cols = 32; int rows = 4096;

            string x = "0.5";
            int y = 13;

            int runs = 3;

            StringBuilder sb = new StringBuilder();
            sb.Append("=1");
            for (int j = 1; j <= y; j++)
            {
                sb.Append("+");
                sb.Append("(");
                for (int i = 1; i <= j; i++)
                {
                    sb.Append(x);
                    if (i < j)
                        sb.Append("*");
                }
            }

```

jul 31, 06 9:17

TaylorNoReferences.cs

Page 2/3

```
                sb.Append(")");
            }
            for (int i = 1; i <= j; i++)
            {
                sb.Append(i);
                if (i < j)
                    sb.Append("*");
            }
            sb.Append(")");
        }
    }

    Workbook wb = new Workbook();
    Sheet sheet = new Sheet(wb, cols, rows);
    sheet.AddCell(sb.ToString(), 0, 0);
    Cell cell = sheet.CopyCell(0, 0);

    for (int col = 0; col < cols; col++)
    {
        for (int row = 0; row < rows; row++)
        {
            if (!(col == 0 && row == 0))
                sheet.PasteCell(cell, col, row);
        }
    }

    Benchmark benchmark = new Benchmark(title, author);

    XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
    GeneratorOptions.UseFormulaSharing = true;

    for (GeneratorLevel level = GeneratorLevel.Level0; level <= GeneratorLevel.Level7; level++)
    {
        // Skip Level1
        if (level == GeneratorLevel.Level1)
            continue;

        GeneratorOptions.Level = level;

        XmlElement data = benchmark.AddData(dataset, level.ToString());
        XmlElement subdata = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
        for (int run = 1; run <= runs; run++)
        {
            watch.Reset();
            watch.Start();
            wb.Recompute();
            watch.Stop();

            benchmark.AddRun(subdata, run, watch.ElapsedMilliseconds.ToString());
        }
        Console.WriteLine("Value: {0}", sheet.ShowValue(0, 0));
    }
    if (filename != null)
    {
        benchmark.Save(filename);
    }
}

```

```
}  
}
```