Speed vs. Memory Usage
-
An Approach to Deal with Contrary Aspects

Wolfgang Schult and Andreas Polze
Hasso-Plattner-Institute
14440 Potsdam, Germany
{wolfgang.schult|andreas.polze}@hpi.uni-potsdam.de

ABSTRACT

Besides design and implementation of components, software engineering for component-based systems has to deal with component integration issues whose impact is not restricted to separate components but rather affects the system as a whole. Aspect-Oriented programming (AOP) addresses those cross-cutting, multi-component concerns. AOP describes system properties and component interactions in terms of so-called aspects. Often, aspects express non-functional component properties, such as resource usage (CPU, memory, network bandwidth), component and object (co-) locations, fault-tolerance, timing behavior, or security settings. Typically, these properties do not manifest in the components' functional interfaces.

Aspects often constrain the design space for a given software system. System designers have to trade off multiple, possibly contradicting aspects affecting a set of components (e.g.; the fault-tolerance aspect may require replication of component data, whereas the security aspect may prohibit it). Component software may be deployed in varying contexts, may be requiring emphasis on only a few of the aspects considered during design and implementation. Static aspect weavers often require compromises with respect to the the generality of services provided by a component system.

Within this paper, we focus on dynamic management of aspect information during program runtime. We introduce a new approach called "dynamic aspect weaving" to interconnect aspect code and functional code. Using our approach, it is possible to decide at runtime whether objects living inside a component should be instantiated with support for a particular aspect or not. We present a distributed Mandelbrot computation as example application and discuss dynamic aspect weaving as technique to manage speed versus memory usage tradeoffs. We have implemented our approach in context of the C# language and the Microsoft .NET environment.

1. INTRODUCTION

There exists a variety of application areas for Aspect-Oriented Programming (AOP). Generally, it is very acceptable to have a preprocessor-like aspect-weaver to interconnect functional code and aspect code. However, sometimes it is desirable to postpone the decision about whether aspect information is to be interwoven with a particular component until program runtime. For instance, one may have a huge resource consuming image processing algorithm located in a component, and depending on system load and available computing nodes a tradeoff between data distribution, memory allocation scheme, and utilization of computing power has to be made at runtime. It might be desirable to distribute calculations for better performance if computing nodes are available. Minimizing local memory usage might be at high priority if the same program is run in a different setting. Both are crosscutting concerns. An aspect may be defined to manage distribution of method invocations across machine boundaries, whereas a different (somewhat contrasting) aspect may deal with local and remote memory utilization during a distributed computation.

Typically, one has to decide at compile time whether an aspect should be interwoven with a set of components or not. Classical AOP techniques provide neither a solution to 'switch off' (ignore) aspect code at runtime nor to dynamically interweave another aspect with the component software.

Within this paper, we present a solution to this problem and demonstrate how to interweave previously defined aspects with functional component code. This 'Dynamic Aspect Weaving' is promising because of its flexibility: neither at design nor at compilation time a definite decision has to be made whether a particular aspect should be applied to a set of components or not. Aspects specialized for a particular situation can be defined and can be interwoven depending on actual runtime requirements. Furthermore one can parameterize the aspects during program runtime. We discuss how this can be accomplished without usage of a special 'aspect weaver' tool.

The remainder of the paper is organized as follows: Section 2 presents related work. Section 3 describes our approach to dynamic aspect weaving. In Section 4 we demonstrate a case study, whose experimental evaluation is presented in Section 5. Section 6 finally summarizes our conclusions.

2. RELATED WORK

The concept of aspect-oriented programming (AOP) offers an interesting alternative for specification of non-functional component properties (such as fault-tolerance properties or timing behavior). There exists a variety of language extensions to deal with AOP. One of which, AspectJ [13], a Java extension, can be cited as the most prominent example. The central concept of most AOP-frameworks is a join point model described in [12][5].

Dynamic join points are an extension of the original AOP model which allows dealing with dynamic informations during runtime [9]. A dynamic joinpoint allows definition of conditions which are evaluated at runtime. Depending on the result, aspect code may be executed or not.

Mehmet Aksit has developed the composition filters object model, which provides control over messages received and sent by an object [3][1]. In this work, the component language follows traditional object-oriented programming techniques, the composition filters mechanism represents an aspect language that can be used to control a number of aspects including synchronization and communication. Most of the weaving happens during runtime.

The authors have implemented a static aspect weaver, which uses the unmanaged metadata interfaces from .NET to interweave aspect code [20].

A restricted technique for dynamic aspect weaving for .NET has been described in [15]. However, this solution uses the current internal debug interfaces of the .NET framework implementation to interweave aspect code during runtime and is therefore less general and portable than our approach.

3. DYNAMIC ASPECT WEAVING

Dynamic aspect weaving means that a component (a *target class*) and an *aspect class* will become interwoven during runtime. There is no need for the aspect class to have a priori knowledge about the target class and vice versa. To understand how the weaving process works, some notions have to be defined.

3.1 What is an Aspect Class?

An aspect description for a set of components focuses on crosscutting concerns. In our case, an aspect is a simple C# class derived from the base class **Aspect**. It will be called *aspect class*. Aspect classes may implement methods, properties, and member variables. In any case, an aspect class describes a way to modify the behavior of another class (the so-called *target class*). Therefore, it is not meaningful to instantiate an aspect class on its own. It rather has to be instantiated jointly with a target class. This process is called *dynamic aspect weaving*. Its technical details will be described later in this section.

3.2 Connection Points

As mentioned above, an aspect class works only in conjunction with an instance of another class. At a *connection point* both will become interwoven. Methods of the aspect class can be identified as connection points, which is indicated by the C# **call** attribute above the method definition in the aspect class. The call attribute is defined as follows:

```
[call(Invoke.InvokeOrder{, Alias=AliasName})]
```

During dynamic aspect weaving, each of the connection points inside an aspect class will become interwoven with a target class' method if at least one of the following requirements is met:

1. The method name and the signature are equivalent.
2. If there is an *AliasName* defined, and the method name from the target class is the same as the alias, and the signatures of both are equivalent.
3. If there is an *AliasName* and the alias contains a wildcard at the end, or the signature of the Aspect class method contains wildcards, and the target method matches.

The following example demonstrates requirement 1:

```
[call(Invoke.Instead)]
void mymethod(int i) { /* ... */ }
```

In this case any target method **mymethod** with one **int** as parameter and **void** as result will interweave with this method in the aspect class.

To demonstrate requirement 2 let us assume that one defines **Alias="myspecialmethod"** for a method. This results in interveaving all target methods named **myspecialmethod** with an **int** parameter and a **void** with the annotated method in the aspect class.

Requirement 3 basically says that if one modifies the alias to **Alias="my*"** every target method beginning with "my" and the same parameters will become interwoven. Furthermore one can use *signature wildcards*. A wildcard for the result type is **object**, and for the parameters **params object[]**, this is like a method with variable arguments. An alias has to be defined in order to flag the argument list **params object[]** as wildcard. The following connection point:

```
[call(Invoke.Instead, Alias="*")]
object catchall(params object[] args)
```

will become interwoven with every method in the target class and *args* will contain each parameter one passes through the method. For instance, if the target class has a method **void f(int i, double d)**, then *args[0]* will contain i and *args[1]* will contain d after the method is called.

Now, since we have described the rules for interweaving connection points with target methods, we are focusing on the actual algorithm implementing dynamic aspect weaving. This is described by the *InvokeOrder* parameter of the call attribute. There are three possibilities:

- **Invoke.Before**: The aspect method of the connection point will be invoked *before* the target method will be called.
- **Invoke.After**: As to be expected, the aspect method will be invoked *after* the target method has been called.
- **Invoke.Instead**: The target method will not be called automatically - but can be called from inside the aspect method.

The first two cases are useful if one wants to trace method calls only. The last case is to be used in order to gain full control over the target method's behavior.

3.3 Aspect Context

When defining an *Invoke.Instead* connection point, one needs a mechanism to call the appropriate target class method.

The problem is that neither the type of the target class (the aspect class can become interwoven with any type) nor, in some cases, the signature of the called method (this is when one uses signature wildcards) are known. The solution is to define a **Context** property in the *Aspect* base class. This property allows access to an object of type **AspectContext** which contains the required information. There are two methods defined for **AspectContexts**:

```
public object Invoke(params object[] args)

public object InvokeOn(object target, params object[]
    args)
```

The first simply invokes the target class' method on an object with the given parameters. The second method allows to invoke the target method on a different, arbitrarily choosen instance (*target*) of the target class. This is useful if there are special instances of the target class stored in the aspect code, and one wants to invoke these.

3.4 Implementation Issues

Within the previous section, we have introduced our notions of an aspect class, of connection points, and of object contexts. Here, we are going to discuss our implementation of those concept. Our approach relies on a number of language features, namely:

- Support of attribute definition.
- Support of reflection to analyse the target class' and the aspect class' signatures (methods and their parameter types).
- Runtime code generation - to emit the interwoven class.

We have implemented our solution based on Microsoft .NET. The Microsoft .NET runtime environment allows to generate, load, and run code on the fly. This code can be presented to the environment in an intermediate language (IL). There exist a variety of programming languages which support .NET and map on the same intermediate language. Since our approach it is possible to interweave an aspect written in one language (say C++), with a component written in a different .NET language (say Pascal).

We have implemented our technique for dynamic aspect weaving in a .NET library. This library provides several classes and attributes defined within the namespace **Aspects**:

- **Aspect** is the base class for all defined aspects.
- **Weaver** is a class which implements the weaving functionality.
- **Call** is an attribute to define connection points.
- **AspectContext** allows invocation of instance methods via *Aspect.Instance*.

3.5 The Dynamic Aspect Weaver

As described above, the **Aspects** namespace contains a class called **Weaver**. It provides special functions to interweave an AspectClass with a specified target class:

```
static object Weaver.CreateInstance(
  Type classtype)
```

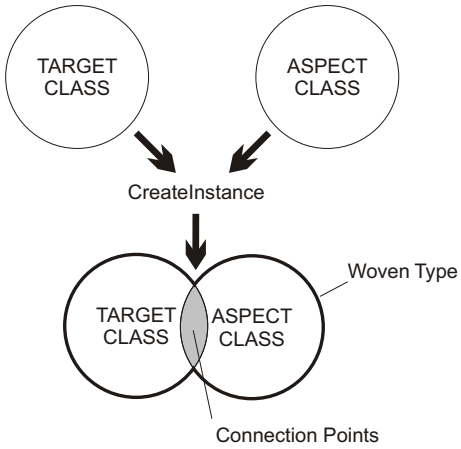

Figure 1: The Weaving Process

```
static object Weaver.CreateInstance(
  Type classtype,
  params object[] args)

static object Weaver.CreateInstance(
  Type classtype,
  params object[] args,
  Aspect aspect)

static object Weaver.CreateInstance(
  Type classtype,
  params object[] args,
  Aspect[] aspectarr)
```

The first and the second version generate an instance of class *classtype*. The objects in *arg* are the constuctor parameters for the target class. The last two versions of *CreateInstance* have an additional parameter *aspect* respectively *aspectarr*, which allows passing of an instance of the AspectClass as argument. A possible call would look like:

```
A a=Weaver.CreateInstance(typeof(A), null, new MyAspect
    ()) as A;
```

In the first two versions of *CreateInstance*, the aspect parameter is given implicitely as an attribute. The following lines have the same effect as the sample above:

```
[MyAspect]
class A
{ /* ... */ }
/* ... */
A a=Weaver.CreateInstance(typeof(A), ...) as A;
```

Giving the aspect instance explicitly as a parameter to *CreateInstance* is more flexible than naming it via attribute - as the aspect and its parameters can be identified at runtime. The code implementing dynamic aspect weaving first looks for a custom attribute derived from **Aspect**. If there is no aspect given, the *CreateInstance* call is equivalent to **new A(*args*)**. What happens during the creation is illustrated in Figure 1. The weaver looks for connection points and tries to join them with the target class' methods as described above. With this information, it builds a new type, and creates a new instance of this type. As final step of a call to *CreateInstance*, the method *Aspect.ctor* will be called. This method can be overriden and has the following signature:

```
virtual void ctor(Weaver weaver, object target, params
    object[] args)
```

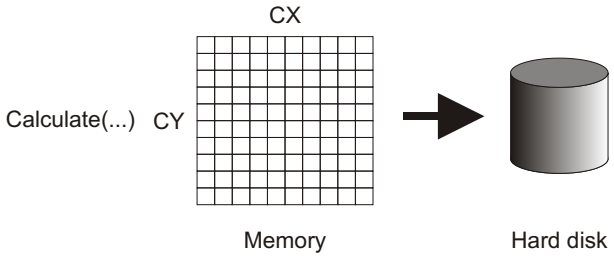

Figure 2: Mandelbrot Function Call

Inside the method, the parameters have the following meaning:

- *weaver* is the aspect weaver itself.
- *target* is the new interwoven instance.
- *args* are the constructor parameters.

Finally, the newly constructed and interwoven object instance will be returned to the caller.

4. CASE STUDY - OPTIMIZING RESOURCE USAGE

Listing 1 shows a C# class which calculates a Mandelbrot set [18]. The input for the algorithm is a filename, a bounding box, and a resolution.

```
public class Mandelbrot
{
  // calculates a given point of the Mandelbrot matrix
  private byte CalculatePoint(double x, double y) { /* ...*/ }

  // only this method is accessible from outside
  // it calculates the matrix and
  // stores the result to the hard disk
  public virtual void Calculate(string filename, double x1,
       double y1, double x2, double y2, int xRes, int yRes)
  {
    double dAddx=(x2-x1)/((double)xRes);
    double dAddy=(y2-y1)/((double)yRes);
    Byte[] matrix=new Byte[yRes*xRes];
    for(int y=0;y<yRes;y++)
    {
      x2=x1;
      for(int x=0;x<xRes;x++)
      {
        matrix[xRes*y+x]=CalculatePoint(x1,y1);
        x1+=dAddx;
      }
      y1+=dAddy;
      x1=x2;
    }
    FileStream fs=new FileStream(filename, FileMode.Create,
         FileAccess.Write);
    fs.Write(matrix,0,matrix.Length);
    fs.Close();
  }
}
```

Listing 1: The Mandelbrot Class

Figure 2 illustrates the behavior of our Mandelbrot computation: The algorithm first calculates the whole Mandelbrot set in memory and then stores it to the hard disk. For small resolutions this is fine. But what happens if the resolution is increased? The amount of memory consumed will increase polynomial (one needs $cx*cy$ memory storage). A possible solution is to rewrite the algorithm. But under certain circumstances, there is not the possibility to do that (i.e. the algorithm exists as binary only), so another solution is needed.

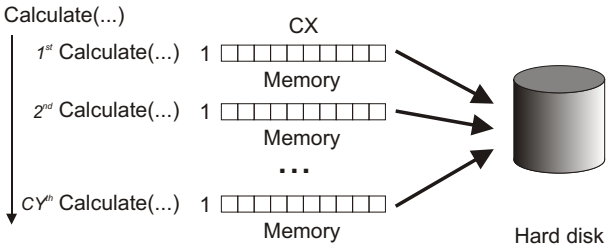

Figure 3: Function Call with the SaveMemory Aspect

4.1 The Save Memory Aspect

The idea is that the function calls are split so that single lines will be processed in memory and subsequently written to separate files on the hard disk. Finally, all these files are joined together to complete the Mandelbrot computation. Figure 2 shows this approach. The envisioned effect can be accomplished using an aspect class (which would not be visible to clients of our Mandelbrot computation). Listing 2 shows a possible implementation of this aspect.

```
public class SaveMemory:Aspect
{
  [Call(Invoke.Instead)] // connection point
  public void Calculate(string filename, double x1, double y1,
       double x2, double y2, int xRes, int yRes)
  {
    // split up in lines
    double dStep=(y2-y1)/((double)yRes);
    for(int i=0;i<yRes;i++)
    {
      // call original function
      Context.Invoke(filename+i.ToString(),x1,y1,x2,y1,xRes,1);
      y1+=dStep;
    }
    // join the files together
    Byte[] data=new Byte[xRes];
    FileStream fsdst=new FileStream(filename, FileMode.Create,
         FileAccess.Write);
    for(int i=0;i<yRes;i++)
    {
      FileStream fssrc=new FileStream(filename+i.ToString(),
           FileMode.Open, FileAccess.Read);
      fssrc.Read(data,0,data.Length);
      fssrc.Close();
      fsdst.Write(data,0,data.Length);
    }
    fsdst.Close();
  }
}
```

Listing 2: The Save Memory Aspect

As visible in the aspect class, the function *calculate* is defined as a connection point. As described in Section 3, if the target class contains a function *Calculate* with the same signature then both will become interwoven. The **for**-loop simply invokes, via the aspect context, the Mandelbrot computation line by line. For n lines it will generate n files on the hard disk. At the end, these n files will become concatenated to form a new file containing the data originally requested.

4.2 The Distribution Aspect

The second goal was to utilize all available processors in a system. Again, we are defining an aspect to tackle this problem. Figure 4 demonstrates what has to be done: First, one instantiates a replica of the original Mandelbrot object on each processor available. Second, on every function call one splits the calculation up and delegates each part to a separate thread. One can use the .NET threadpool for this.

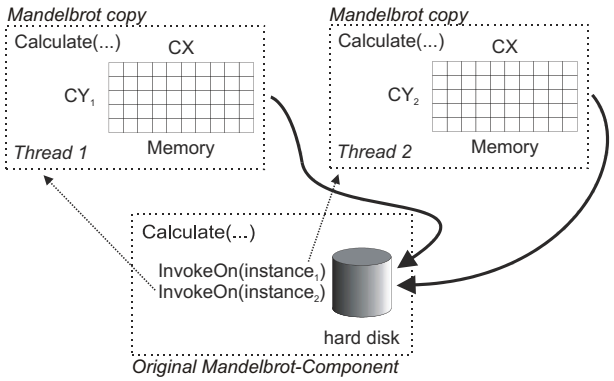

Figure 4: Function Call with the Distribution Aspect

The original Mandelbrot object gets the results back from each replica and joins them together. Listing 3 shows an excerpt of the actual C# implementation.

```
public class Distribute:Aspect
{
  private object[] instances;
  private int workcount;

/* ... */

  // here we generate the copies from the mandelbrot component
  public override void ctor(Weaver weaver, object o, object[]
       args)
  {
    // get processor count from current system
    System.Int32 affinity=System.Diagnostics.Process.
         GetCurrentProcess().ProcessorAffinity.ToInt32();
    int iInstances=0;
    while(affinity!=0)
    {
      if((affinity & 1)!=0) iInstances++;
      affinity=affinity>>1;
    }
    // and generate copies
    instances=new Object[iInstances];
    while(iInstances--!=0)
    {
      instances[iInstances]=weaver.CreateInstance(o,args);
    }
  }
  // the connection point
  [Call(Invoke.Instead)]
  public void Calculate(string filename, double x1, double y1,
       double x2, double y2, int xRes, int yRes)
  {
    // split up calculation in threads
    workcount=instances.Length;
    int nyRes=yRes/workcount;
    double yStep=(y2-y1)/((double)yRes);
    double yRange=yStep*nyRes;
    AutoResetEvent ev=new AutoResetEvent(false);
    double ny1=y1;
    int iNum;
    for(iNum=0;iNum<instances.Length-1;iNum++)
    {
      double ny2=ny1+yRange;
      System.Threading.ThreadPool.QueueUserWorkItem(
        new WaitCallback(Distribute.InvokeWorker),
        new WorkItem(       // this is a container for
          this,             // aspect
          ev,               // event
          instances[iNum],  // mandelbrot instance
          GetFilename(iNum), // temporary filename
          x1, ny1, x2, ny2, // boundaries
          xRes, nyRes));    // resolution
      ny1=ny2+yStep;
    }
    System.Threading.ThreadPool.QueueUserWorkItem(
      new WaitCallback(Distribute.InvokeWorker),
      new WorkItem(this, ev, instances[iNum],GetFilename(iNum),
          x1, ny1, x2, y2, xRes, yRes-(nyRes*(instances.Length
          -1))) );
    // wait until ready
    while(workcount!=0) ev.WaitOne();
    // join files
    FileStream fsdst=new FileStream(filename, FileMode.Create,
         FileAccess.Write);
    for(iNum=0;iNum<instances.Length;iNum++)
      Copy(GetFilename(iNum),fsdst); // copy file to filestream
    fsdst.Close();
  }
  // callback for threadpool
  public static void InvokeWorker(object para)
  {
    // unpack parameters from workitem and start calculation
    WorkItem item=(WorkItem)para;
    item.aspect.Context.InvokeOn(item.target, item.filename,
         item.x1, item.y1, item.x2, item.y2, item.xRes, item.
         yRes);
    // signal ready
    Interlocked.Decrement(ref item.aspect.workcount);
    item.readyevent.Set();
  }
}
```

Listing 3: The Distribution Aspect (excerpt)

The aspect class contains three important functions. The first is **ctor**, which will be called by the Weaver when the instance is created. It is used to create additional instances of the same type which may process function calls in parallel. The second is **Calculate**. This method contains the **call** attribute, which defines it as connection point as well. Here the function calls are executed in separate threads operating on disjunct copies of the Mandelbrot object.

4.3 The Client Side

On the client side, only the instantiation of the Mandelbrot class changes. Depending on the actual runtime environment, one or the other aspect will become interwoven with the Mandelbrot class (Listing 4).

```
Mandelbrot mb;
// we need less memory usage
if(opt_memory.Checked)
  mb=Aspects.Weaver.CreateInstance(typeof(Mandelbrot),null,new
       SaveMemory()) as Mandelbrot;
// we need more performance
else if(opt_speed.Checked)
  mb=Aspects.Weaver.CreateInstance(typeof(Mandelbrot),null,new
       Distribute("d:/temp")) as Mandelbrot;
// we need nothing of both
else mb=new Mandelbrot();
```

Listing 4: The Client Side

The function call initiating the actual Mandelbrot computation does not change.

5. PERFORMANCE MEASUREMENTS

After implementing a dynamic weaver and designing two aspects, we evaluate the performance impact of our dynamic aspect weaver. For our experiments, we have used a 1GHz Dual-Pentium III System with 256MB RAM. The first measurement evaluates object instantiation time. We have created objects with one to five member functions, comparing the behavior of our dynamic aspect weaver (invoked via *CreateInstance*) with the standard **new** operator's behavior. The aspect itself defines a connection point which will interweave with every function in the target class. Furthermore we have repeated each instantiation several times in a test run. The whole test cycle has been repeated 300 times for each category to obtain statistically relevant data.

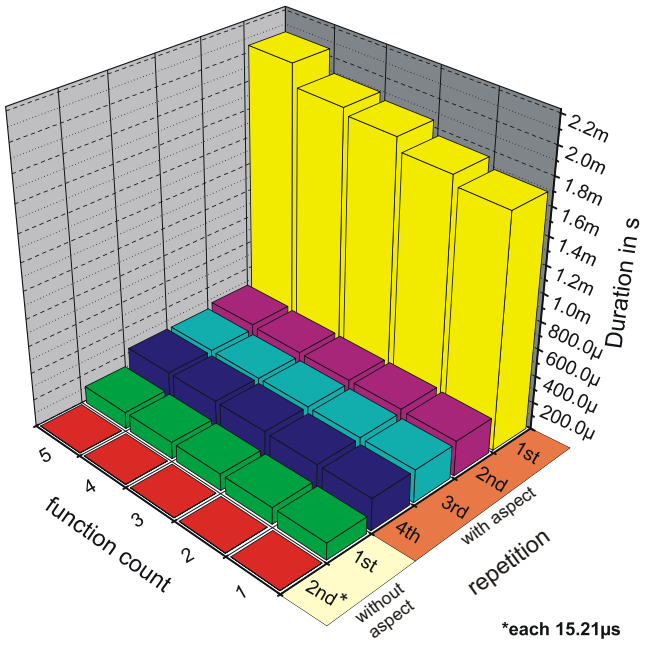

Figure 5: Average duration (300 measurements per bar) of object instantiation with the aspect weaver and with "new"

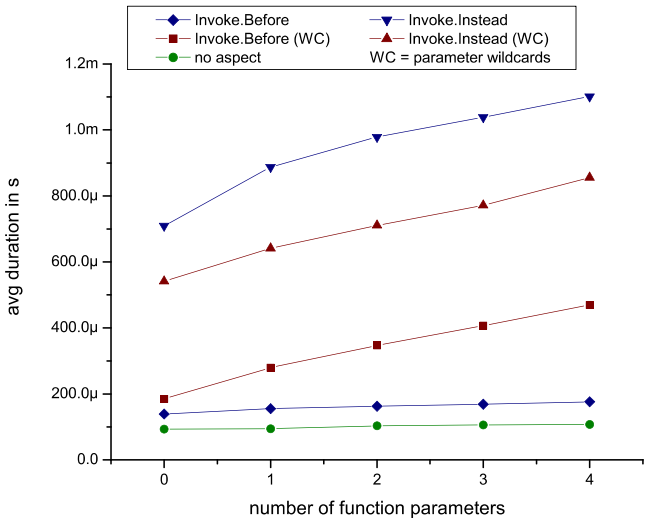

Figure 6: Average duration (300 measurements per point) of a first function call with several types of connection points in a function mix

Figure 5 presents the result of our measurements. The columns show the complexity of the object (in this case the number of declared member functions). The first two rows belong to instantiations without an aspect (with the **new** statement). The last four rows are instantiations using our dynamic aspect weaver. The duration of the initial object instantiationis correlated with the complexity of the object. This is because for every function that exists in the target class, the weaver has to generate a connection code between aspect and target class. The second instantiation is much faster (approximately ten times faster) than the first one. Here the weaver can use the formerly generated code to instantiate a new object. Therefore, the duration does not depend on the object complexity (function count). The third and fourth instantiations are a bit faster than the second instantiation. In this case the intermediate language code (IL-code) has been fully processed by the just-in-time compiler and only native code will be executed. Comparing the instantiations with an aspect to the instantiations without an aspect, one can see that the first instantiation without an aspect is approximatly 20 times faster than with. Later on, object instantiation with our dynamic aspect weaver is by a factor or 12 slower than without aspect weaving.

In our second measurement we want to show the overhead of function calls with a varying parameter counts and the timing differences caused by the various types of connection points being used. As a test environment we used a function mix which consisted of 12 assignments, 8 control flow statements, and 4 function calls. Every 4th function call is interwoven with aspect code. In Figure 6 we show the results for the first call and Figure 7 shows the first repetition of each function call. First of all, one recognizes that each second call is much faster than its first occurance. Again, this effect is caused by the just-in-time compiler of the .NET runtime environment. But more important is the discovery that only

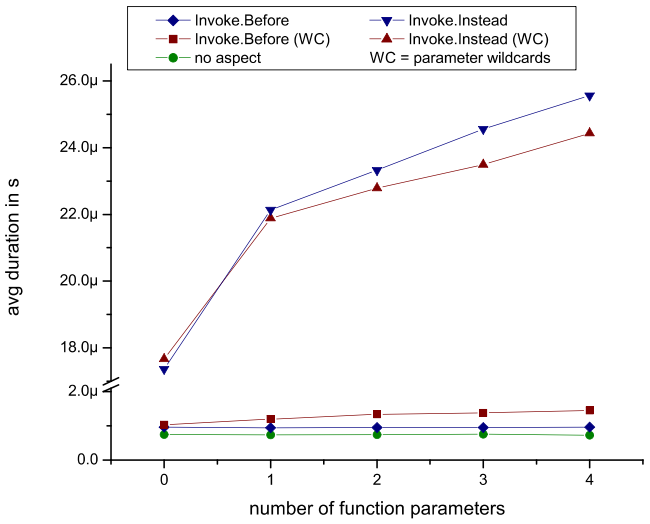

Figure 7: Average duration (300 measurements per point) of a second function call with several types of connection points in a function mix

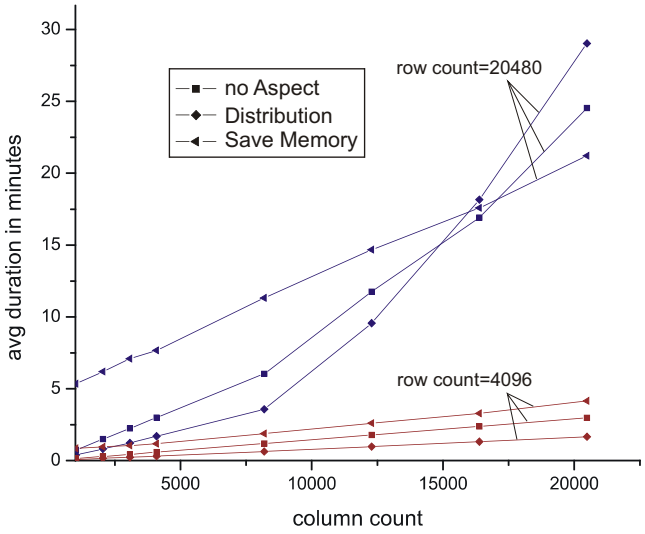

Figure 8: Comparison of average duration (20 measurements per point) between both aspects and without any aspect in the mandelbrot component

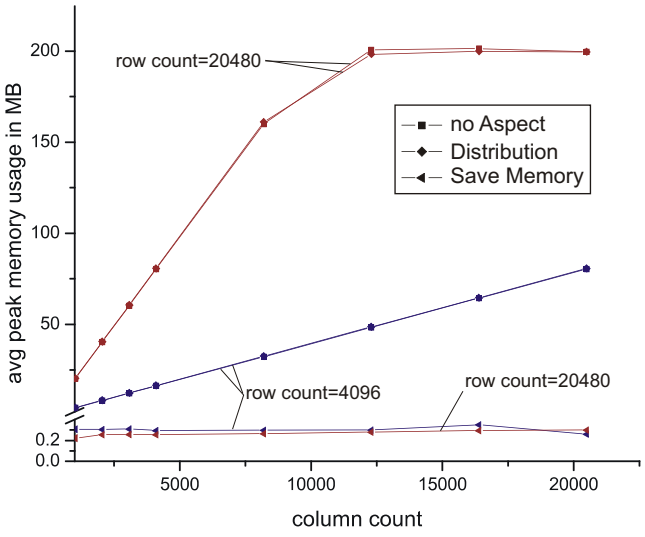

Figure 9: Comparison of average peak memory usage (20 measurements per point) between both aspects and without any aspect in the mandelbrot component

Invoke.Before connection points and calls without an aspect have nearly constant execution times (the influence of submitted parameters is barely measurable). In the three other cases the execution time increases with the number of parameters. In this case, special code will be executed, which deal with the different signatures of the target class function and the function declared as a connection point. The last point one sees is that **Invoke.Instead** connection points take much longer (approximately 20 times longer) than **Invoke.Before** connection points. This phenomena is explained by the late binding of the target class function calls through the Aspect Context.

Back to our example, we show the impact of both, the Distribution and the Save Memory Aspect on our system resources. Figure 8 shows the average duration of the mandelbrot calculation in dependence on the number of calculated columns in the mandelbrot matrix (CX). We sketched out two representive row counts (CY) for the three cases. For a row count of 4096 one can see that the algorithm with the Distribution Aspect assigned is approximately twice as fast as it is without an aspect. With a Save Memory aspect we have a performance gap of approximately fifty percent compared to the calculation without an aspect. But with a row count of 20480 the situation changes markedly.

Beginning at a column count of approximately 17200 the algorithm assigned the Save Memory aspect gets the best performance. The explanation for that is shown in Figure 9. One sees that in the measurements with a column count 12288 the maximum of available physical memory has been exhausted. On the other hand, the Algorithm with the Save memory aspect assigned uses a consistently low amount of memory. This prevents it from swapping out memory and so decreasing its performance.

6. CONCLUSIONS

Aspect-oriented programming (AOP) is a relatively new approach for separation of concerns in software development. AOP makes it possible to modularize crosscutting aspects of a system.

We have presented our approach to dynamic management of aspect information at program runtime. We have introduced a novel technique called "dynamic aspect weaving" which allows for late binding (weaving) of aspect code and functional code. Using our approach, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment. Relying on the .NET support for a variety of programming languages, our approach is not restricted to C#, but works for all of the .NET languages.

Our current implementation has some constraints for the programmer of a component. Currently, only virtual methods can be interwoven dynamically. The reason for this lies in our implementation of late binding of the function calls. Currently the Weaver "overrides" the function so that the virtual method table maintained inside the .NET virtual machine points to the woven function (the version enriched with aspect information). Other members of a class, such as fields, properties, static, and class functions currently cannot be accessed this way. However, recursively applying the AOP techniques described here and in [20], it is a simple task to generate proxy classes which substitute non virtual member functions and fields with their virtual counterparts.

7. REFERENCES

[1] M. Aksit and L. Bergmans. Composing multible concerns using composition filters. *Communications of the ACM*, 44, Issue 10:51–57, Oktober 2001.

[2] M. Aksit and B. Tekinerdogan. Aspect-oriented programming using composition-filters. In *ECOOP'98 Workshop Reader*. Springer Verlag, 1998.

[3] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP'98 workshop position paper, 1998.

[4] T. Archer. *Inside C#*. Microsoft Press, 1 edition, 2001.

[5] AspectJ Homepage. http://www.aspectj.org/, 2002.

[6] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 86–95, Enschede, The Netherlands, April 22-26 2002. ACM press.

[7] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. In *Communications of the ACM*, volume 44, pages 33–38, Oktober 2001.

[8] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. In *Communications of the ACM*, volume 44, pages 30–32, Oktober 2001.

[9] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 21-22 2002.

[10] S. Hanenberg and R. Unland. Concerning aop and inheritance. In *Aspektorientierung - Workshop der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung*, Paderborn, Germany, May 3-4 2001.

[11] S. Hanenberg and R. Unland. A proposal for classifying tangeled code. In *Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 21-22 2002.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44, Issue 10:59–65, October 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Finnland, June 1997. Springer Verlag LNCS 1241.

[14] J. O. K.Lieberherr, D. Orleans. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44, Issue 10:39–41, Oktober 2001.

[15] J. Lam. My runtime aspect weaver. http://www.iunknown.com, 2002.

[16] C. V. Lopes and G. Kiczales. *Recent Developments in AspectJ*. Xerox Palo Alto Research Center.

[17] D. Mahrenholz, O. Spinczyk, and W. Schrder-Preikschat. Program instrumentation for debugging and monitoring with aspect c++. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 249–256, Crystal City, VA, USA, April 29 - May 1 2002.

[18] B. Mandelbrot. *The Fractal Geometry of Nature*. Freeman, San Francisco, 1982.

[19] Microsoft Cooperation, http://msdn.microsoft.com/net/ecma/. *ECMA C# and Common Language Infrastructure Standards*, 2001.

[20] W. Schult and A. Polze. Aspect-oriented programming with C# and .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.

[21] W. Schult and A. Polze. Dynamic aspect-weaving with .NET. In *Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen*, TU Berlin, Germany, November 7-8 2002.