ISSN 1801-2108 *Volume 4, Number 1, 2006*

Journal of .NET Technologies

An international journal on software components, large-scale software, software correctness and security, object-oriented techniques, programming paradigms, multi-language programming, multithreading and distributed applications, high-performance computing, web services and virtual machines, algorithms, data structures and techniques, human computer interfaces with .NET, related projects with .NET, educational software and teaching object-oriented paradigms with .NET

EDITOR - IN - CHIEF

Václav Skala
University of West Bohemia

Journal of .NET Technologies

Editor-in-Chief: Vaclav Skala
University of West Bohemia, Univerzitni 8, Box 314
306 14 Plzen
Czech Republic
skala@kiv.zcu.cz
Managing Editor: Vaclav Skala

Author Service Department & Distribution:
Vaclav Skala - UNION Agency
Na Mazinach 9
322 00 Plzen
Czech Republic
Reg.No. (ICO) 416 82 459

Hardcopy: ***ISSN 1801-2108***
ISBN 80-86943-13-5

4th .NET Technologies 2006

University of West Bohemia
Campus Bory

May 29 – June 1, 2006

Co-Chair

Jens Knoop, Vienna University of Technology, Austria
Vaclav Skala, University of West Bohemia, Czech Republic

CONFERENCE CO-CHAIR

Knoop, Jens (Vienna University of Technology, Vienna, Austria)
Skala, Vaclav (University of West Bohemia, Plzen, Czech Republic)

PROGRAMME COMMITTEE

Aksit, Mehmet (University of Twente, The Netherlands)
Giuseppe, Attardi (University of Pisa, Italy)
Gough, John (Queensland University of Technology, Australia)
Huisman, Marieke (INRIA Sophia Antipolis, France)
Knoop, Jens (Vienna University of Technology, Austria)
Lengauer, Christian (University of Passau, Germany)
Lewis, Brian,T. (Intel Corp., USA)
Meijer, Erik (Microsoft, USA)
Ortin, Francisco (University of Oviedo, Spain)
Safonov, Vladimir (St. Petersburg University, Russia)
Scholz, Bernhard (The University of Sydney, Australia)
Siegemund, Frank (European Microsoft Innovation Center, Germany)
Skala, Vaclav (University of West Bohemia, Czech Republic)
Srisa-an, Witawas (University of Nebraska-Lincoln, USA)
Sturm, Peter (University of Trier, Germany)
Sullivan, Kevin (University of Virginia, USA)
van den Brand, Mark (Technical University of Eindhoven, The Netherlands)
Veiga, Luis (INESC-ID, Portugal)
Watkins, Damien (Microsoft Research, U.K.)

REVIEWING BOARD

Alvarez, Dario (Spain)
Attardi, Giuseppe (Italy)
Baer, Philipp (Germany)
Bilicki, Vilmos (Hungary)
Bishop, Judith (South Africa)
Buckley, Alex (U.K.)
Burgstaller,Bernd (Australia)
Cisternino, Antonio (Italy)
Colombo, Diego (Italy)
Comito, Carmela (Italy)
Ertl, Anton,M. (Austria)
Faber, Peter (Germany)
Geihs, Kurt (Germany)
Gough, John (Australia)
Groesslinger, Armin (Germany)
Huisman, Marieke (France)
Knoop, Jens (Austria)
Kratz, Hans (Germany)
Kumar,C., Sujit (India)
Latour, Louis (USA)
Lewis, Brian (USA)
Meijer, Erik (USA)
Midkiff, Sam (USA)
Ortin, Francisco (Spain)
Palmisano, Ignazio (Italy)
Pearce, David (New Zealand)
Piessens, Frank (Belgium)
Safonov, Vladimir (Russia)
Schaefer, Stefans (Australia)
Scholz, Bernhard (Australia)
Schordan, Markus (Austria)
Siegmund, Frank (USA)
Srinkant, Y.N. (India)
Srisa-an, Witawas (USA)
Strein, Dennis (Germany)
Sturm, Peter (Germany)
Sullivan, Kevin (USA)
Tobies, Stephan (USA)
van den Brand, Mark (The Netherlands)
Vaswani, Kapil (India)
Veiga, Luis (Portugal)

Contents

- Aaltonen,A., Buckley,A., Eisenbach,S.: Flexible Dynamic Linking for .NET (United Kingdom) 1
- Zychla,W.: eXtensible Multi Security: Contracts for .NET Platform (Poland) 9
- Kühn,E., Fessl,G., Schmied,F.: Aspect-Oriented Programming with Runtime-Generated Subclass Proxies and .NET Dynamic Methods (Austria) 17
- Ertl,M.A., Thalinger,Ch., Krall,A.: Superinstructions and Replication in the Cacao JVM Interpreter (Austria) 25

Flexible Dynamic Linking for .NET

Anders Aaltonen, Alex Buckley, Susan Eisenbach
a.buckley@imperial.ac.uk
Imperial College London

Abstract

A .NET application is a set of assemblies developed or reused by programmers, and tested together for correctness and performance. Each assembly's references to other assemblies are type-checked at compile-time and embedded into the executable image, from where they guide the dynamic linking process.

We propose that an application can potentially consist of multiple sets of assemblies, all known to the application's programmers. Each set implements the application's functionality in some special way, e.g. using only patent-free algorithms or being optimised for 64-bit processors. Depending on the assemblies available on a user's machine, the dynamic linking process will select a suitable set and load assemblies from it.

We describe how, in our scheme, an application is written to use a *default* set of assemblies but carries nominal and structural specifications about permissible sets of *alternative* assemblies. We implement the scheme on Rotor, a .NET virtual machine, by modifying its linking infrastructure to efficiently find assemblies on the user's machine that satisfy the application's specifications. Specifications can be applied to individual classes and methods, so that only code wishing to use alternative assemblies has to undergo the modified linking process.

1 Introduction

1.1 Dynamic linking in .NET

Modern virtual machines, like the Common Language Runtime in .NET, support dynamic linking of bytecode obtained from local and remote sites. The key concept in .NET linking is the *assembly*. An assembly is a file that contains classes' bytecode and serves as a versioned, tamper-proof unit of deployment. To guide linking, an assembly has metadata that describes its classes' dependencies on other assemblies and *their* classes. Source languages typically disallow a programmer from specifying which assembly is to provide which class; the choice is made by the compiler.

.NET Technologies 2006

So, for a method call in C♯:

```
System.Console.WriteLine(‘‘Hello’’)
```

the compiler will choose an assembly in the compile-time environment that contains the `System.Console` class, e.g. `mscorlib` 1.0.5. The compiler embeds the name and version of the chosen assembly into the metadata of the assembly being built, and emits bytecode that references (in `[...]`) the chosen assembly's name:

```
ldstr ‘‘Hello’’
call void [mscorlib]System.Console::WriteLine(string)
```

.NET will (try to) link exactly the version of the `mscorlib` assembly specified in the executing assembly's metadata. This helps to avoid "DLL hell" [6], because the user's machine can have multiple versions of an assembly installed, *e.g.* `mscorlib` 1.0.5 for application A and `mscorlib` 1.1.0 for application B, and both application's dependencies can be satisfied.

The problem is that while an assembly specified in the metadata of an executing assembly *was* available at compile-time on the programmer's machine, the user's

machine may *now* have alternative assemblies available at run-time. Reasons why assemblies at run-time may differ from those at compile-time include:

- .NET's standard libraries provide interfaces for well-understood features like XML processing, database access and networking, so it is straightforward for multiple vendors to provide different implementations of the interfaces. Each vendor's assembly is likely to have a different name.
- Within a company's IT department, developers often have different implementations of a business interface that version numbering alone cannot reasonably differentiate. For example, two assemblies that contain different implementation classes for a bond trading strategy may well be signed by different keypairs and have different versioning conventions; and thus different names.

Unfortunately, linking in .NET cannot cope when assemblies in the run-time environment have different names to those in the compile-time environment. The .NET assembly loader can only redirect a request for one *version* of an assembly to another version of the same assembly; it cannot redirect the *name* of an assembly. Thus, a programmer who wishes to make his application portable between differently-named assemblies (that are expected to contain implementation classes for popular interfaces) must code portability by hand. Typically, a Factory pattern or an inversion-of-control container [9] is used to abstract class names from the main application code, but some reflective, type-unsafe code is always needed to discover assemblies and extract implementing classes.

1.2 Flexible dynamic linking in .NET

We propose a more declarative approach to portability. An application programmer merely enumerates assemblies and classes that his application can use (*e.g.* that provide implementation classes of useful interfaces), and the dynamic linker finds and instantiates them as available. Thus, we reduce an assembly's dependence on a particular assembly (known to a compiler) by adding *potential* dependencies that increase the range of valid run-time environments.

In our scheme, *any* assembly or class name that appears in bytecode can be redirected. The programmer writes code as usual that references classes, but includes *nominal specifications* along the lines of "try assemblies `B` and `C` as well as `A`" and "try classes `P` and `Q` as well as `R`". After a compiler has generated an assembly from the programmer's code, we have a tool that, for the purpose of type-safety, takes the assembly and adds *structural specifications* based on its nominal specifications and the classes and members that it references. For example, given the nominal specifications above, the generated structural specifications would be along the lines that "any assembly used in place of assembly `A` must provide class `D`" and "any class used in place of `D` must provide a field `f` of type `E`".

Our modified dynamic linker inspects an assembly's nominal and structural specifications at run-time.[1] If an assembly name referenced in bytecode is not available, then the linker searches for substitute assemblies given in the nominal specifications; any found assembly must satisfy the structural specifications. Then, when an assembly's classloader searches for classes, it considers the nominal and structural specifications for classes.

Even with structural specifications providing type-safety, it is unlikely that an assembly exists at run-time with the "right" classes (with the "right" members) *unless the programmer knew about it in advance*. This is because only the programmer can ensure that the assemblies and classes named in nominal specifications are semantically compatible with (*i.e.* exhibit the same observable behaviour as) assemblies and classes known to the compiler. Thus, our policy is that only assemblies directly referenced in bytecode or enumerated in nominal specifications should be linked. To prevent third parties making or modifying specifications, specifications are embedded in an assembly's metadata rather than being expressed in standalone resource files that anyone could edit.[2]

1.3 Related work

The use of type variables for abstracting over data types is well-known in the functional [12] and object-oriented [10] worlds. For example, rather than having bytecode refer to specific classes, introducing type variables at bytecode at compile-time can provide true sep-

[1] Strictly speaking, the linker works at JIT-time, loading assemblies in support of member resolution. But we consider JIT-time as "run-time", because after bytecode is JIT-compiled, it is extremely difficult to inspect which assemblies and classes it uses.

[2] We assume that assemblies will routinely be strongly-named, thus making them tamper-proof. This is analogous to how a publisher policy is a strongly-named assembly that contains an XML file, rather than a standalone XML file.

arate compilation for object-oriented languages [1]. In [3], we also advocated inserting type variables into bytecode at compile-time and substituting them to available assembly and class names at run-time. We modified the .NET dynamic linker to recognise type variables, but the end-user could specify substitute assemblies and classes without any guarantee that their substitutions were type-safe, so clearly the system was not realistic.

Most work on assemblies in .NET concerns a coherent relationship between executing assemblies and installed assemblies. [7] describes a management tool that can, by respecting a model of binary compatibility, configure a program to safely use a different version of an assembly. *Type forwarders* [11] are a feature in .NET 2.0 that allow a class to be moved from one assembly to another without breaking programs that reference the class in its original assembly. Metadata is added to the class's old assembly specifying a new assembly, and Fusion silently redirects all requests for the old assembly to the new assembly. The feature is needed by framework maintainers because, as noted earlier, Fusion cannot redirect assembly names nor does it deal with classes. Type forwarders' redirections are one-to-one and unknown to the programmer, whereas our redirections are one-to-many and intended for programmers and deployers.

As the scope of link-time activity grows, describing the behaviour of dynamic linking gains importance. Dynamic linking for Java was formalised [8, 4] because of its perceived complexity. In fact, Java's linking for unversioned, unsigned classes is considerably simpler than .NET's linking for versioned, signed assemblies, and [2] describes the assembly resolution and loading process for various .NET implementations. [5] provides a simple framework for linking in both the Java Virtual Machine and .NET.

1.4 Structure of this paper

§2 describes the high-level features available to a programmer in our system for making code more flexible with respect to its execution environment. §3 describes the architecture of a dynamic linker capable of choosing assemblies and classes at run-time, and explains a key abstraction, the *LinkContext*. §4 describes our extensions to the dynamic linker of "Rotor", the shared-source version of Microsoft's .NET Framework.

2 Design

2.1 Specifying flexible linking

To let a programmer specify alternative assembly and class names, we define two classes of custom attribute. Custom attributes are a mechanism in .NET for specifying non-functional program properties in a language-independent way. They are attached to source language constructs, such as classes and methods in an object-oriented language, and have a canonical representation in bytecode.

Our custom attributes are `[LinkAssembly]` and `[LinkClass]`; we call them *linking attributes*. In fig. 1, we assume that class `C` uses GUI classes - specifically `System.Windows.Forms` - supplied with .NET on Windows. To help `C` run on a .NET implementation on MacOS or Linux, where `System.Windows.Forms` *may* exist but where alternative assemblies providing GUI classes *may* be available, we attach linking attributes to specify both the alternative assemblies and their classes.

```
[LinkAssembly(‘‘System.Windows.Forms’’,
              ‘‘cocoa’’, ‘‘1.3.*’’,
              ‘‘macos’’, LOCAL_INTERFACE)]
[LinkAssembly(‘‘System.Windows.Forms’’,
              ‘‘qt’’, ‘‘*’’,
              ‘‘linux’’, LOCAL_INTERFACE)]
[LinkClass(‘‘System.Windows.Forms.Button’’,
           ‘‘GelButton’’,
           ‘‘macos’’)]
[LinkClass(‘‘System.Windows.Forms.Button’’,
           ‘‘qButton’’,
           ‘‘linux’’)]
class C { ... }
```

Fig. 1: Preparing code for flexible linking

Attributes are attached to the `C` class to specify that any reference in `C`'s bytecode to the assembly `System.Windows.Forms` can be redirected by the dynamic linker to either 1) an assembly `cocoa` of version 1.3.x that the programmer expects to be available on MacOS, or 2) an assembly `qt` of any version ("*") that the programmer likes to use on Linux.

If either of these redirections happens, then class names used in `C` will be redirected by the dynamic linker. The `GelButton` class will be used in preference to `System.Windows.Forms.Button` if the linker chose assembly `cocoa`, while `qButton` will be used if the linker chose assembly `qt`.

We say that `C`'s bytecode is *subject to flexible linking* since it is in the scope of at least one `[LinkAssembly]` attribute. Our modified .NET dynamic linker will recog-

nise where code is subject to flexible linking, while an unmodified linker will ignore any linking attributes and simply link bytecode to the types embedded in metadata by the compiler.

2.2 Semantic interfaces

If the code above happens to run on a Linux machine, then the likelihood is that only the `qt` assembly and not MacOS' `cocoa` assembly would be found. It therefore makes sense to only look for Linux-specific assemblies after finding `qt`. To capture the fact that different `[LinkAssembly]` attributes are likely related by platform, vendor or maturity (*e.g.* alpha, beta, production), the penultimate parameter of `[LinkAssembly]` is a *semantic interface name* that characterises the relation. If a `[LinkAssembly]` attribute specifies an assembly name that is actually used by the dynamic linker, then we support multiple policies for which linking attributes to consider in future. The policy is determined by the final parameter of the successful `[LinkAssembly]`, which is called its *semantic interface qualifier* and can take one of the following values:

LOCAL_INTERFACE If an assembly has already been chosen based on a `[LinkAssembly]` with this semantic interface qualifier, then all further assembly and class resolutions in the same scope must use linking attributes with the same semantic interface name as that `[LinkAssembly]`.

LOCAL_INTERFACE_PREFERRED If an assembly has already been chosen based on a `[LinkAssembly]` with this semantic interface qualifier, then `[LinkAssembly]` attributes with the same semantic interface name are checked first when resolving other assemblies and classes in the same scope. If none of these attributes successfully specify a loadable assembly, then other `[LinkAssembly]` attributes are tried.

LOCAL_INTERFACE_EAGER A `[LinkAssembly]` attribute with this semantic interface qualifier is "eager" in the sense that all `[LinkAssembly]` attributes in the same scope with the same semantic interface name must be successfully resolved immediately.

ANY_INTERFACE No restriction on later resolutions.

2.3 Attribute Scoping

Custom attributes can be attached to assemblies, modules, classes and methods. This aids expressiveness because an attribute can be attached to the most refined scope necessary; only methods that require flexibility need to have attributes. We search for attributes "inside out" to aid performance, *i.e.* first at the method level, then the class and assembly levels.

As an example of how attribute scoping works, the following code calls `List::op1` and `List::op2` on an `ArrayList` implementation of a `List` interface. But if possible, the programmer would like to use the `SinglyLinkedList` implementation in the assembly that encloses class `C`, because `List::op1`'s traversal is suited to a linked list rather than an array-based list. `m2`, however, calls `List::op2`, that can reasonably expected to traverse the list backwards as well as forwards, so a `DoublyLinkedList` would be helpful:

```
[assembly: LinkClass(ArrayList, SinglyLinkedList)]

class C {
  void m1() {
    List l = new ArrayList();   l.op1();
  }

  [LinkClass(ArrayList, DoublyLinkedList)]
  void m2() {
    List l = new ArrayList();   l.op2();
  }
}
```

Linking attributes apply not only to member references, but to any type within an attribute's scope. Thus, the following code permits an object of either class `C` or `D` (or any of their subclasses) to be passed to `m1`, and an object of either class `C` or `E` (or any subclass) to `m2`.

```
[LinkClass(C,D)]
class C {
  void m1(C x) { ... }

  [LinkClass(C,E)]
  void m2(C x) { ... }
}
```

2.4 Type safety

Assemblies and classes specified in linking attributes must be binary-compatible with the assemblies and classes referenced by bytecode, or else resolution exceptions (*i.e.* "message not understood" errors) could arise at run-time. We therefore need a way to ensure that any assembly/class specified in a linking attribute chosen by the linker is type-compatible with

all references to the original assembly/class throughout an assembly. An assembly's metadata enumerates which other assemblies and classes it depends on, the members accessed in those classes are found only in individual bytecode instructions. Hence, they are only revealed at JIT-compilation when each instruction in the assembly is verified. To avoid an extra pass over bytecode during JIT-compilation, we gather constraints about member accesses with a compile-time tool, and store them as custom attributes attached to methods. These *constraint attributes* are similar in style to those in [13] and [1].

A `[LinkMemberConstraint]` attribute describes required fields and methods of classes, *e.g.*

```
[LinkMemberConstraint(``A1'', ``C1'', 100, ``M1'')]
```

states that whatever class is linked for `[A1]C1` is expected to contain a member (field or method signature) defined by token 100 in module `M1`. (A module is a unit inside an assembly that actually holds the assembly's class definitions. The metadata for the classes' dependencies - on other assemblies, class and members - is stored at the module level rather than class level, and indexed by integers known as tokens.)

A `[LinkSubtypeConstraint]` attribute encapsulates subtype constraints, *e.g.*

```
[LinkSubtypeConstraint(``A1'', ``C1'', 100,
                       ``A1'', ``C2'', 200, ``M1'')]
```

states that whatever type replaces `[A1]C1` is a supertype of whatever replaces `[A1]C2`. (100 and 200 are the tokens where `[A1]C1` and `[A2]C2` are defined in metadata.)

Fig. 2 shows how source code is annotated with linking attributes to support flexible dynamic linking. Ideally, a .NET compiler would emit member and subtype constraints after successful type-checking. But, to stay language-independent, we built a small program, `flex`, that inspects an assembly's bytecode, identifies member accesses and inserts `[LinkMemberConstraint]` attributes at the appropriate scope. Unfortunately, we cannot generate subtype constraints without performing complex data-flow analysis, as the verifier does during JIT-compilation. We currently require a programmer to specify `[LinkSubtypeConstraint]` attributes manually.

3 Architecture

We now describe how flexible dynamic linking is architected in Microsoft's shared-source version of .NET known as "Rotor". There are two candidates for which run-time subsystem should perform flexible linking for members and types: 1) the resolver called by the JIT-compiler, or 2) the loaders called by the resolver to physically find assemblies on disk and extract classes from them. The latter is an attractive place to check linking attributes, because .NET's assembly loader already consults user-defined policies for redirecting assembly versions. But, if the redirection to load a different assembly/class is done at too low a level and not exposed to the higher-level resolver, then the wrong types may be loaded later in the resolution process. For example, our constraint verifier needs to know exactly what assemblies and classes have been loaded in order to check member and class definitions. Therefore, we prefer to place our implementation closer to the JIT-compiler's resolver. (We do not consider performance, but do not believe either subsystem would have an advantage.)

Fig. 3 summarises where our implementation (boxes with bold text) lives in Rotor. It sits just below the the high-level resolution algorithm, intercepting requests to resolve members and types, and modifies requests those before assemblies and classes are actually loaded. By sitting just above the assembly loader, we apply linking attributes to a member or type resolution before user-defined policies are applied. This is appropriate, because versioning policies, *e.g.* to avoid a security flaw in an old assembly, should apply even to assemblies named in linking attributes.

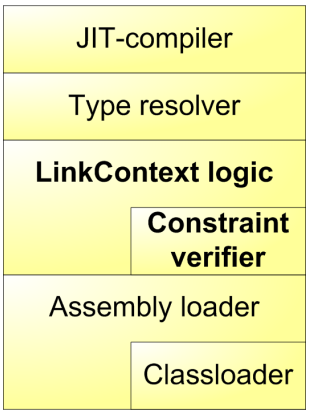

Fig. 3: Overview of flexible dynamic linking in Rotor

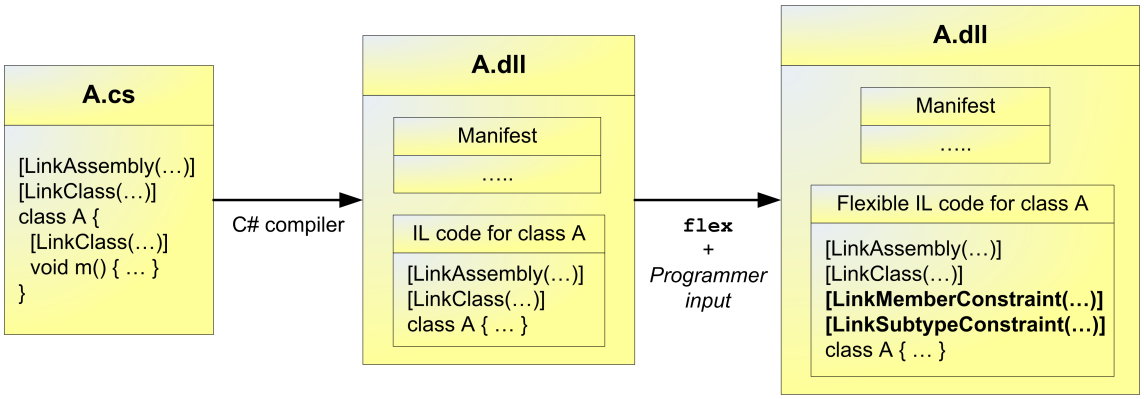

Fig. 2: Preparing code for flexible linking

3.1 Linking contexts

A member access (field access or method call) instruction in bytecode contains an integer "token" that is mapped, in metadata, to a *member descriptor* very similar to a field or method signature, *e.g.* `[A]C::m(void)`. Resolution is the process during JIT-compilation that turns a token, via a member descriptor, into a first-class object that directly represents the member's definition in a loaded class from a loaded assembly. Similarly, a class declaration contains one or more tokens that map to *type descriptors* for its superclasses, *e.g.* `[A]C`. Also, a type-cast instruction contains a token for the target type. Resolution turns these tokens into objects representing loaded class definitions. Note that a token always references at least an assembly name and a class name.

When resolving a particular token, we need to consider the linking attributes and constraint attributes applicable to, *i.e.* in scope at, the token being resolved. We call the set of in-scope linking attributes applicable to a token its *resolving context*. Each token has its own resolving context because different linking and constraint attributes apply to it.

We introduce a *LinkContext* to compute and encapsulate a resolving context. For a particular token, a LinkContext finds all the `[LinkAssembly]` attributes declared closest to it. If previous resolutions chose `[LinkAssembly]` attributes whose semantic interface qualifier was **LOCAL_INTERFACE_PREFERRED** or **LOCAL_INTERFACE_EAGER**, then a LinkContext will find only those `[LinkAssembly]` attributes with the appropriate semantic interface names. The resolving context consists of those `[LinkAssembly]` attributes, plus `[LinkClass]` attributes (with the appropriate semantic interface name, if necessary) in the same scope as the `[LinkAssembly]` attributes, plus constraint attributes in the same scope. A LinkContext can be queried for the linking and constraint attributes "relevant" to a particular token, *e.g.* if the token being resolved is for a type `[A]C`, then only `[LinkAssembly]` attributes for assembly `A` are relevant.

4 Implementation

4.1 Modifying the JIT-compiler

We add a stack of LinkContexts to each module loaded from an assembly. When a method is JIT-compiled, we push a "master" LinkContext on to the module's stack for efficiency reasons. This LinkContext immediately gathers all the linking and constraint attributes (at method, class, module and assembly levels) in scope for the method. These attributes are a superset of any individual token's resolving context.

Whenever the JIT-compiler reaches a token that it needs resolved, we push a further LinkContext on to the module's stack (and pop it after the token has been resolved). This LinkContext computes the token's resolving context by querying the master LinkContext for attributes in scope for the token, then selecting appropriate linking and constraint attributes as described in §3.1.

To actually push a LinkContext when the JIT-compiler encounters an unresolved token that refers to a member or type, we modify methods called by the JIT-compiler that resolve a token: `CEEInfo::findField`, `CEEInfo::findMethod` and

`CEEInfo::findClass`. These methods are made to create and destroy LinkContexts as follows:

```
// If linking attributes present...
if (pLink->HasLinkContext() && pLink->IsScopeFlexLinked())
  // Create a nested LinkContext
  pLink->NewNestedLinkContext(...);
else
  pLink = NULL;

// Pre-existing resolution code to
// find a field/method/class
...

if (pLink)
  // Remove the LinkContext from the stack
  pLink->GetParentLinkContext();
```

4.2 Modifying assembly loading

The pre-existing resolution code that we have elided above calls the assembly loader to visit the filesystem. As usual, the loader looks up the assembly name (of the token being resolved) in the currently executing assembly's metadata. This gives various details, such as the version number of the token's referenced assembly, which are stored in an `AssemblySpec` object. At this point, our code intercedes, passing the `AssemblySpec` to the top LinkContext on the current module's stack.

The LinkContext uses the "master" LinkContext to build the resolving context for the current token, then picks just the `[LinkAssembly]` attribute that specifies a redirection for the assembly mentioned by the token. (If there is more than one possible `[LinkAssembly]`, we pick the first.) For example, if the token mentions assembly `A`, then having `[LinkAssembly(‘‘A’’, ‘‘B’’, ‘‘1.0’’...]` in the resolving context will cause the LinkContext to choose assembly `B` v1.0. The LinkContext then loads this substitute assembly and performs some security checks that will be performed by the JIT-compiler later; we do not wish its checks to fail.

Having chosen and loaded a substitute assembly, we update the `AssemblySpec` object with the substitute assembly's name and pass the object back to the usual assembly loading logic. Since the assembly has already been loaded by LinkContext for constraint verification, it will be found immediately in the assembly loader's cache.

4.3 Modifying class loading

Ordinarily, once a valid assembly is loaded, the JIT-compiler's pre-existing resolution code uses the assembly's classloader to load the token's class. We intercede in the classloader to ask the top LinkContext on the current module's stack to choose a substitute class based on the resolving context.

The LinkContext again uses the "master" LinkContext, this time to retrieve a `[LinkClass]` attribute in the token's resolving context that has the appropriate semantic interface name and is for the token's class. The LinkContext tries to load the class specified in the `[LinkClass]`, and verify any applicable constraint attributes for it.

To respect `[LinkMemberConstraint]` attributes, a LinkContext first uses the ordinary method and field resolvers `EEClass::FindMethod` and `EEClass::FindField` to check the presence of members in the substitute class's definition (an `EEClass` object). Then, it verifies that the signatures of the members requested in constraint attributes match exactly the signature of the member found in the class. This entails resolving and loading each type (*i.e.* assembly+class) in the found members' signatures, such as method formal parameters. Since those members are in classes that *themselves* may have linking attributes, further LinkContexts are created and the whole flexible dynamic linking process recurses. A similar issue arises when verifying subtypes to respect `[LinkSubtypeConstraint]` attributes.

Having checked constraints on substituted classes, we pre-empt a later check by the JIT-compiler, which is that any loaded class is visible to the method being JIT-compiled. If the substitute class is visible and its definition satisfies member and subtype constraints, then the class's member definition or the class definition itself (depending on whether the token is a member descriptor or a type descriptor, respectively) is cached by the LinkContext for that token. The substitute class's name is then used by the classloader to retrieve an `EEClass` class definition from the assembly, as usual, and this succeeds immediately since we already loaded the class's definition to check constraints.

4.4 Source language issues

When compiling a method call, Rotor's C♯ compiler statically binds to the class that defines the method, relying on the runtime to dynamically dispatch the method in a subclass if necessary. Consider the following C♯ source code:

```
class A     { virtual  void m1() { ... } }
class B : A { override void m1() { ... } }

// Main program
[LinkClass(B,...)]
{ ... new B().m1(); ... }
```

The compiler produces bytecode that specifies `m1` *in class* `A`:

```
[LinkClass(B,...)]
newobj   instance void [...]B::.ctor()
callvirt instance void [...]A::m1()
```

At runtime, the body of `B::m1` will be executed as usual. But if the programmer wrote `[LinkClass]` attributes with alternatives to class `B`, they will never be used. LinkContext only sees a call (the `callvirt` instruction) to a method in class A, which no `[LinkClass]` attribute mentions. We could partially fix the problem by modifying the compiler to bind to the *virtual* declaration of `m1` rather than the overriding declaration; Sun made this change to `javac` between JDK1.3 and JDK1.4. Modifications to the virtual machine would also be necessary.

5 Conclusion

Dynamic linking in .NET is over-constrained because it must provide exactly the types known to a compiler on a programmer's machine. While software engineering techniques can find and link alternative code at run-time, they have to be coded into each application and often use type-unsafe reflection. We have designed a scheme that lets the programmer describe alternative choices for what types can be linked, which is the only way to ensure observational equivalence with types named in source code. If our dynamic linker picks a different type from that named in source code, then any check for type-safety, security or class visibility will succeed if it would have succeeeded for the original type. .NET's ability to attach attributes to code allows for precise specification of what and where choices should be available in a program, in a way that causes no overhead to unmodified .NET virtual machines. Also, our specifications let the programmer reflect the fact that families of assemblies are often grouped together logically, *e.g.* patent-free algorithms, so that linking one assembly should restrict later linking to the same family.

Further work is identifying when to perform flexible linking even if a compiler has "hidden" the opportunity with its static resolution, and finding real-world applications that can benefit from our scheme. Increasing portability between mobile and desktop frameworks may be a fruitful avenue, particularly as the number grows of .NET-enabled mobile devices with API support for differentiated capabilities (GPS, wi-fi, cameras, etc).

Acknowledgements This work was partially funded by a Microsoft Research grant under the 2004 Rotor RFP II. We thank Sophia Drossopoulou for comments.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, Long Beach, CA, USA, January 2005.

[2] Alex Buckley. A Model of Dynamic Binding in .NET. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2005)*, Glasgow, Scotland, July 2005.

[3] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. Flexible Bytecode for Linking in .NET. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005)*, ENTCS, Edinburgh, Scotland, March 2005. Elsevier BV.

[4] Sophia Drossopoulou. An Abstract Model of Java Dynamic Linking and Loading. In Robert Harper, editor, *Proceedings of the Third International Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 53–84. Springer-Verlag, 2000.

[5] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

[6] Susan Eisenbach, Vladimir Jurisic, and Chris Sadler. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002. `http://joint.org/use2002/`.

[7] Susan Eisenbach, Dilek Kayhan, and Chris Sadler. Keeping Control of Reusable Components. In *Proceedings of Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.

[8] T. Jensen, D. Le Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 4–15, Chicago, IL, USA, 1998.

[9] Rod Johnson. The Spring Framework. `http://www.springframework.org/`, 2005.

[10] Andrew Kennedy and Don Syme. The Design and Implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, Snowbird, UT, USA, June 2001.

[11] Richard Lander. The Wonders of Whidbey Factoring Features. `http://hoser.lander.ca/`, 2005.

[12] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL'93)*, pages 439–450, Charleston, SC, USA, 1993.

[13] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2003)*, Anaheim, CA, USA, October 2003.

eXtensible Multi Security: Contracts for .NET Platform

Wiktor Zychla
University of Wroclaw, Poland
wzychla@ii.uni.wroc.pl

Abstract

This paper presents XMS – a language independent security framework called *eXtensible Multi Security* which is designed to verify that modules written in .NET languages are safe with respect to *Contracts* Safety Policy. Dynamic verification engine uses code instrumentation to supervise the execution and validate contracts at run-time. Static verification engine is based on the Proof-Carrying Code paradigm where it is up to the Code Producer to construct a *Certificate of Safety* – formal logic proof enclosed in the code – which can be used by a Code Consumer to verify that the code is secure.

Keywords: intermediate language, dynamic verification, static verification, contracts

1 INTRODUCTION

Distributed systems play a major role in today's computer systems. However, the convenience and freedom offered by such systems is sometimes misused. The software and the hardware is a potential victim to a malicious virus, the data is a potential victim to a trojan horse or a spy-software.

In fact, there is a lot of carelessness when dealing with distributed systems. There are critical bugs found even in vital parts of Operating Systems and commonly used applications. It is still very easy to trick the trusting user and make him run a malicious code on his system and it is usually impossible to check the software and decide if it is secure or not.

Alas, over forty years after the Internet has been born, the majority of users still have to believe that the software they buy or download is secure in the sense that it will not do any harm to their hardware and data.

Widely spread antivirus software can detect several thousands of computer viruses. That's good. Alas, it is able to detect only these viruses that are known. That's bad. If a new virus is released, my machine is probably vulnerable again.

Runtime environments can dynamically supervise the code execution and disallow the execution of some potentially harmful activities. That's good. They cannot however make sure that the code runs correctly. That's bad. Even the fancy managed code is not a bit helpful when the banking software steals money from my bank account.

The goal of eXtensible Multi Security (XMS) is to unify various ideas in one coherent and extensible platform. XMS evolved from logic systems that form a powerful certification framework based on a notion of Proof Carrying Code (PCC, [11]).

The original PCC approach focuses on type-safety. However, the type-safety does not guarantee that other important aspects of safety are preserved. In fact, various aspects of security are rather independent. For example, the code can be type-safe but not correct or type-unsafe but perfectly safe from the control flow point of view.

This is where the XMS starts. XMS infrastructure focuses on selected notions of security and applies them to the existing Microsoft Intermediate Language (MSIL) Runtime Environment. XMS is designed in the spirit of .NET platform – digital certificates are **language independent**. Certificates are put in **attributes** and then stored in binary **meta-data** so that they do not play any role in the code execution but instead they can be used in the verification process.

XMS is a Work in Progress – currently about **60%** from over 200 MSIL opcodes are supported by Static Contracts certification tools. Since compilers of some high-level languages use only selected subsets of MSIL opcodes, XMS is yet compatible with some existing high-level compilers, for example the C# compiler.

1.1 Static vs Dynamic Security

In *dynamic* checking, the safety policy is constantly checked at the run time. This of course requires the existence of a virtual machine or a runtime environment that would be powerful enough, in the sense that it could detect any activity that breaks the safety policy. An example of such an infrastructure is the Java Virtual Machine or the Microsoft .NET Framework. Both "supervise" the execution of code, and enforce precise

.NET Technologies 2006

checks before any potentially dangerous instruction is executed.

On the other hand in the process of *static* checking the code is verified without being actually run. The answer of a static check is always positive or negative, and the code is accepted or rejected. It is impossible to break the execution in the middle, as in the dynamic approach. A static checking does not necessarily need any support from a runtime environment. In this sense it is more general than the other. However, the static security policies are usually less precise because all non-trivial security policies are undecidable. The user must then accept the fact that some programs would be misjudged which means that some perfectly legal programs could be rejected (the opposite situation, where an illegal code was accepted, would be a true disaster and should never happen).

All these observations lead to an obvious conclusion: there is no perfect way to enforce a safety policy. The best what we could probably do would be to put the advantages of dynamic and static checking together in a framework that unifies all advantages of these two.

1.2 XMS = Static + Dynamic Security

This is exactly what XMS is. On one hand, XMS is built to certify .NET code and that is why .NET dynamic security policies are still validated in run-time. On the other hand, XMS is built on the top of the Proof Carrying Code paradigm and that is why the safety policies are verified statically.

Here is a short summary of XMS benefits:

- XMS is designed to certify the MSIL language, one of the most promising and widely used intermediate languages.
- XMS certificates are compatible with high-level .NET languages. A high-level language developer **does not need to know** MSIL to certify the code.
- To support XMS the .NET Runtime Environment **does not need to be changed** in any way.
- XMS certificates are built around the notion of PCC thus inherit all desirable properties of PCC:
  - the certificates are sufficient to guarantee that the code is valid,
  - the authority of a code producer is completely insignificant to the code security.

2 COMPARISON TO RELATED WORK

Formal verification of software has long history ([15]). The PCC framework ([11]) was a milestone at this area. PCC was proposed to certify the type-safety of low level languages as the alternative to the TAL ([14]). There are several main PCC research directions:

- exploiting the core of PCC paradigm ([5])
- applying PCC type-safety to industrial environments ([9])
- developing other safety policies for research languages ([1], [2])

For many reasons the type security is strongly desirable for assembly-level languages. In such approach the primary goal of PCC is to validate the language compiler by detecting compile-time bugs. This idea was further adopted to certify the type safety of Java binaries at machine-level (SpecialJ compiler described in [9]).

Initially XMS started as a PCC variant for a toy-like object language. After migration to .NET platform, XMS marks out its own way:

- XMS does not certify type-safety of the low level language but instead it allows to certify other safety policies of the MSIL language.
- Since the certificates can be applied to any high-level language, XMS is more general than solutions bound to a single low-level ([11]) or high-level ([6]) language.
- XMS will ultimately adopt other security policies, such as Non-Interference, to its verification engine

Currently, as a contract verification framework, XMS competes with specialized contract frameworks for .NET Platform like the Spec# ([20]). Major differences between these two:

- Unlike XMS, Spec# is bound to a single language - it is a superset of C#.
- Unlike XMS, Spec# is bound to a single safety policy (contracts). XMS is an extensible framework with pluggable verification engines
- In Spec# contracts are declared using the language extensions and turned into inlined code during the compilation. In XMS, contracts are external to the language (attributes) and code instrumentation techniques are used for dynamic analysis
- Spec# uses its own intermediate representation of the code, BoogiePL, which is interpreted and transformed before it is provided to the theorem prover. XMS uses symbolic evaluation to build verification traces directly from the .NET Intermediate Language code.

3 XMS CERTIFICATES

3.1 Certification Scheme

The PCC certification scheme is based on **Verification Conditions**, logic predicates that contain information

about the execution of programs. XMS Verification Conditions are built by the **VCGen** – a tool that scans MSIL binaries and performs symbolic evaluation of the MSIL code. The Theorem of XMS Safety (3.1) says that **if** certificates are provable **then** properties of corresponding programs hold. Thus, formal proofs of Verification Conditions can be used as **digital certificates**. Such certificates are unbreakable since it is impossible to hack a formal logic system if it is proven to be sound and correct.

The XMS certification protocol assumes that a Safety Policy is shared between Code Producer and Code Consumer. The protocol is shown in Figure 3.

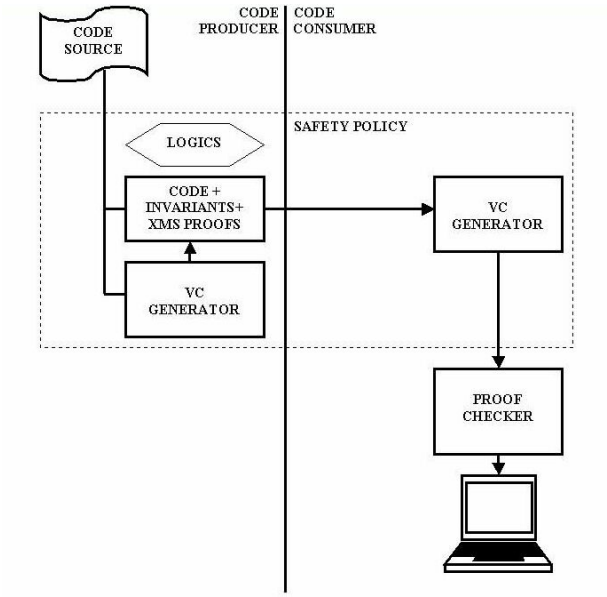

Figure 1: XMS Certification Protocol

The Code Producer and Code Consumer use the same public and verified safety policy that define logic and algorithms to build Verification Conditions for each logic.

The Code Producer:

1. adds method specifications to the source code,
2. uses VCGen to build and encode Verification Conditions (VC),
3. constructs proofs for VCs,
4. embeds VCs and proofs as a metadata (metadata is not used at runtime but is extracted in the certification process).

The Code Consumer:

1. uses VCGen to build Verification Conditions,
2. checks if the same VCs have been supplied with the code by the Code Producer,
3. validates the correctness of proofs (certificates).

Note that the protocol can fail at some point at the Code Consumer side. Specifically:

1. the MSIL binary does not contain the metadata that is required to build Verification Conditions,
2. the predicates built at Code Consumer side can differ from these supplied with the code,
3. proofs supplied with the code can be invalid in the sense that they do not prove Verification Conditions.

If the protocol **fails** for any of these reasons the Code Consumer should **reject** the code as unsafe.

XMS introduces the concept of **Verification Traces**. While Verification Condition is a predicate that captures any execution of a method, the Verification Trace is a predicate that represents execution of a single trace of a method. And while Verification Condition acts like a digital certificate which verifies code correctness, Verification Traces can be used by developers to identify possible invalid execution traces in the code - any Verification Trace that is not provable refers such possible invalid execution sequence.

3.2 Contracts

The **Design By Contracts** paradigm lays the base for systematic object-oriented development [7]. It defines a precise framework where software components can be seen as communicating entities whose interaction is based on mutual obligations. These obligations take the form of predicates: preconditions, postconditions and invariants. It means that the specification of each method must be a quadruple:

$$Spec_F = (Sig_F, Pre_F, Post_F, Inv_F)$$

where Sig_F is a method's signature, Pre_F is a precondition predicate, $Post_F$ is a postcondition predicate, Inv_F is a partial function that maps MSIL instruction numbers to invariants.

Currently XMS supports Eiffel-style Contracts [8] with the complete compatibility (i.a. subcontracting) as the ultimate goal. Contracts are provided in attributes.

3.3 Dynamic Contracts

There are two main techniques of code instrumentation for the .NET platform, .NET Profiler API and context-bound objects. .NET Profiler API is a great way for transparent instrumentation since is it completely decoupled from the source code. It is however COM-based and thus not portable. For now XMS uses then context-bound objects and by implementing `IContributeServerContextSink` interface it is able to intercept method invocations and returns.

3.4 Static Contracts

Both Security Policy for static verification and accompanying Theorem are stated formally using formal operational semantics of the .NET intermediate language (defined for XMS formally by following the .NET draft [21]).

Definition 3.1 (Safety Policy of Contracts). *A method F is safe with respect to Static Contracts if for any initial state $\Sigma_0 = (0, \rho_0)$ such that $\rho_0(Pre_F)$ and any state $\Sigma = (i, \rho)$ reachable from the initial state we have that if F_i =* `ret` *then $\rho(Post_F)$. We will denote this fact as $Safe_{SC}(F)$.*

$$Safe_{SC}(F) \iff$$

$$\forall_{\Sigma_0=(0,\rho_0),\Sigma=(i,\rho)}\ \rho_0(Pre_F) \wedge \Sigma_0 \mapsto^* \Sigma \wedge F_i = \texttt{ret} \Rightarrow$$

$$\rho(Post_F)$$

A module $\mathscr{M}$ is safe with respect to Static Contracts if all methods from the module are safe. We will denote this fact as $Safe_{SC}(\mathscr{M})$.

$$Safe_{SC}(\mathscr{M}) \iff \forall_{F \in \mathscr{M}}\, Safe_{SC}(F)$$

This formal definition gives the base of the XMS Static Contracts certification. It says that if a method's precondition is satisfied when the execution begins then the method is safe only if the postcondition is satisfied when the execution is about to end. It also says that Static Contracts are *modular* which means that a module is safe only if all its methods are safe.

The underneath theorem is the central part of Static Contracts for XMS. It formally states the soundness of the VC-based certification framework.

Theorem 3.1 (Theorem of Safety for Static Contracts). *If the verification condition for a given module M is valid, i.e. if $\models VC(\mathscr{M})$ then all executions of any module methods are correct with respect to their contracts, i.e. $Safe_{SC}(\mathscr{M})$.*

The VCGen algorithm the theorem refers to and the proof of the theorem are long, technical and will not be presented here. Both are inductive on the MSIL instruction set.

Symbolic evaluation of an object language (like MSIL) which is a heart of the algorithm arise several issues:

arithmetics an arithmetic instruction causes VCGen to update its symbolic store to new state.

conditionals a conditional jump causes VCGen to split the symbolic evaluation into recursive paths for all branches. Conditions became assumptions inside the verification predicate.

backward jumps backward jumps could lead to infinite analysis. VCGen requires then that each backward jump targets instructions which have *invariants* provided. Invariants are validated when they are seen for the first time and then validated again when a backward jump is encountered.

method calls a method call makes VCGen to put the method's precondition as an assumption into the predicate and then initialize a new state with all variables which could be modified inside the called method (out parameters) set to new, fresh values.

objects objects are evaluated symbolically.

arrays an array is stored as a index-value dictionary.

polymorphism is it not known until the run-time which exact method is called from a class hierarchy. VCGen relies here on a *subcontracting* paradigm ([8]) according to which contracts of inherited methods must depend on contracts of base-class methods.

0**-values** contracts must allow to use original values in postconditions. VCGen uses special form of an assumption to support such possibility.

3.5 XMS Architecture

XMS Architecture is presented in Figure 2. Both engines (static and dynamic) are written in C#.

The main core of the dynamic verification engine is about 250 lines long and uses .NET context attributes and message sinks to instrument the code at run-time. Expressions are evaluated using .NET dynamic code execution technique.

The main core of the static verification engine has currently 1500 lines of code but uses external parser for specification parsing and external IL decompiler. The symbolic evaluator maintains the state of evaluated code between recursive calls and produces either one Verification Condition or a set of Verification Traces for each method. A simple windowed user interface is provided for user's convenience.

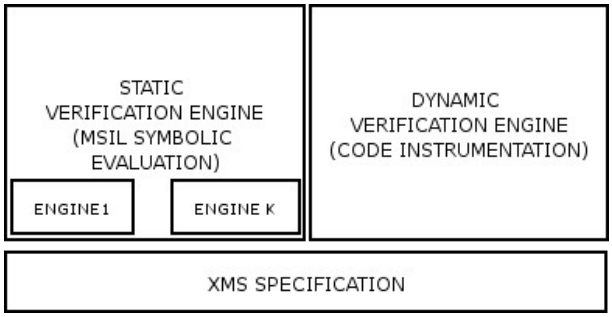

Figure 2: XMS Architecture

3.6 An Example of XMS Certification

To give a glimpse of the XMS framework we present a simple example of an interactive session with the XMS toolkit.

Suppose that the Code Producer has a code of a simple C# method to compute the GCD of two positive integer numbers using the Euclid algorithm:

```
[XMS_Spec(
  "x>=0 & y>=0", "VALUE = GCD(x,y)",
  "GCD(x,y)=GCD(V_0,V_1)", "by auto.")]
public static int GDC( int x, int y ) {
  int k = x;
  int l = y;

  while ( k-l != 0 ) {
    if ( k > l )
      k -= l;
    else
      l -= k;
  }

  return k;
}
```

There is nothing special with the code except for the `XMS_Spec` attribute. It is supplied by the Code Producer and it carries the information about the method's specification:

- the precondition: $x \geq 0 \wedge y \geq 0$
- the postcondition: $VALUE = GCD(x,y)$
- the loop invariant: $GDC(x,y) = GDC(V_0,V_1)$

The precondition establishes an initial assumption on method's parameters. The postcondition characterize method's expected behaviour. The invariant describe a constant condition that is valid in any execution of the loop.

The Verification Condition Generator (VCGen) produces the Verification Condition that captures all essential aspects of arbitrary invocation of the method. Both Code Producer and Code Consumer use VCGen invoked on a binary module:

```
> VCGen.exe gdc.exe
```

It examines the module structure, reads the MSIL binary code and specification metadata and produces the Verification Condition. Note how the specification and conditional branches become assumptions for further parts of the VC.

```
forall x. forall y. (x >= 0 & y >= 0 =>
 (((x-y) =0=> x = GCD(x,y)) &
  ((x-y)!=0=>GCD(x,y)=GCD(x,y) &
     forall V_0_. forall V_1_.
       GCD(x,y)=GCD(V_0_,V_1_)=>
         ((V_0_>V_1_ =>
          ((((V_0_-V_1_)-V_1_) =0=>
             (V_0_-V_1_) = GCD(x,y)) &
           (((V_0_-V_1_)-V_1_)!=0=>
              GCD(x,y)=
              GCD((V_0_-V_1_),V_1_)))) &
          (V_0_<=V_1_ =>
           (((V_0_-(V_1_-V_0_)) =0=>
           V_0_ = GCD(x,y)) &
            ((V_0_-(V_1_-V_0_))!=0=>
            GCD(x,y)=
            GCD(V_0_,(V_1_-V_0_)))))))))
```

The last XMS attribute parameter, "`by auto.`" is the proof of the predicate supplied by the Code Producer. In our example this is a simple proof for a tactical theorem prover. The Code Consumer uses this proof to verify the reliability of the C# method – because the proof is correct for the Verification Condition ($\models VC(F)$), the Code Consumer can be sure that *any* execution of the method is safe according to the Static Contracts Safety Policy ($Safe_{SC}(F)$). Ultimately, XMS will allow to use a tactical theorem prover (Isabelle) to shorten proofs or a proof checker (Twelf) for faster validation.

Another example:

```
[XMS_Spec( 0,
  "n >= 0",
  "VALUE=sum(0, n)",
  "V_0=sum(0, V_1) & n >= V_1",
  "", "")]
public int Sum_I( int n )
{
  int sum = 0;
  for ( int k=0; k<=n; k++ )
  {
    sum += k;
  }

  return sum;
  }
```

Produces following Verification Condition:

```
forall n. (n >= 0 => 0=sum(0, 0) & n >= 0 &
  forall V_0_. forall V_1_.
  V_0_=sum(0, V_1_) &
  n >= V_1_=>
    ((V_1_<n => (V_0_+(V_1_+1))=
      sum(0, (V_1_+1)) & n >= (V_1_+1)) &
     (V_1_>=n => V_0_=V_0_ &
       V_0_=sum(0, n))))
```

Let us also look at the example of dynamic verification:

```
[XMSIntercept]
public class Test : ContextBoundObject
{
```

```
  [Process(typeof(XMSProcessor))]
  [XMS_Spec( 1,
    "true",
    "x == y_0 && y == x_0", "", "", "" )]
  public void Swap( ref int x,
                    ref int y )
  {
    int z = x;
    x = y;
    y = z;
  }
}
```

This time the method is declared to swap input values, the return value of x is equal to original value of y (y_0) and vice versa. Actual client code:

```
int u = 0, v = 1;
t.Swap( ref u, ref v );
```

And the engine outputs:

```
Preprocessing Test.Swap.
Specification found:
Pre=[true]
Post=[x == y_0 && y == x_0]

Precondition : true
Substituted expression : true
Evaluated expression : True

Postcondition : x == y_0 && y == x_0
Substituted expression : 1 == 1 && 0 == 0
Evaluated expression : True
```

4 FROM MSIL TO HIGH-LEVEL LANGUAGES

The .NET paradigm unifies many programming languages at MSIL level. Whether you use C#, VB.NET, Managed C++ or any other .NET language, your code can closely cooperate with any other .NET code.

Since the Verification Condition Generator works at MSIL level, it cannot determine which language was used to produce MSIL binary. And no matter if a binary was produced by IlAsm compiler, C# compiler or any other language compiler, it should be certifiable in the uniform way.

The goal of "lifting" the certification framework from MSIL to a high-level language is then executed under two paradigms:

- A high-level language developer should not be forced to learn MSIL language. In particular, a solution where a high-level code is first compiled to MSIL and then manually certified is unacceptable. Certificates should be then easily applicable to a high-level language code.
- A high-level compiler should not require any major changes to support the certification. In fact, it would be perfect, if the high-level compiler did not require **any** changes. In particular, existing high-level language compilers should not damage certificates that were applied to high-level code.

It seems that comparing to other security policies, Static Contracts is quite difficult to be lifted to high-level languages. There are several important difficulties that have to be addressed:

- Static Contracts Invariants have the form $Inv_F(i) = (P, \ldots)$ where i is the MSIL instruction number and P is the invariant predicate. It could be however extremely difficult to determine the MSIL instruction number for given high-level instruction, since it would require a deep knowledge of compiler transformation routines.
- During the compilation to MSIL, names of local variables are omitted.

The first difficulty can be addressed with a clever technical trick. We would like to avoid attributing instruction numbers to invariant predicates. We would rather like to have an ordered set of invariants:

$$Invs_F = (P_0, \ldots, P_n)$$

and somehow infer Inv_F from it by mapping consecutive invariants to instructions that need invariants.

This goal can be achieved with additional scan of the binary code which could discover instructions $I = (i_0, \ldots, i_k)$ that are targets for backward jumps.

We could then take:

$$Inv_F(i) = \begin{cases} P_j & \text{if } i = i_j \text{ for some } j \text{ and } j \leq n \\ \varepsilon & \text{in other case} \end{cases}$$

The second difficulty can be addressed by "virtually" renaming consecutive local variables to $v_0, \ldots, v_n$ and using these "virtual" names in specifications by a high-level language developer.

4.1 Common Certificate Specification

Both above technical tricks require that the high-level language satisfies two important conditions. These conditions are **essential** for the "lifting" process to work, so we will formulate them as the **Common Certificate Specification** (by analogy to Common Language Specification and Common Type System, two fundamental .NET paradigms). The Common Certificate Specification is as follows:

Variable Ordering Consecutive high-level language local variables become consecutive MSIL local variables.

Structure of Loops High level language loops become MSIL loops with as simple structure as possible.

While the above specification does not look formal enough, we are not going to make it formal. It is because some important existing compilers (like the C# compiler) fulfill both these requirements, so the CCS formulation should be treated as a set of guidelines for new compilers.

Both requirements are crucial for proper translation of loop invariants between a high-level language and MSIL. In an example from section 3.6 the loop invariant refers to variables `k` and `l` but in MSIL they become V_0 and V_1. Since there is only one loop in C# code, only one loop invariant should be supplied. VCGen will automatically detect instructions which invariants refer to.

In fact, the main reason that makes the "lifting" possible is that .NET high-level language compilers follow few simple and obvious patterns while producing MSIL from high-level code. This is not a coincidence and chances are that future compilers will also behave in similar way because MSIL is not a platform-native language – it is the Just-In-Time compiler which does most of fancy optimizations while translating the MSIL to platform-native language.

Of course this "simple translation with obvious patterns" rule applies mainly to C# and VB.NET, two main business languages for the .NET platform. Other languages with non-trivial translation schemes must find their own way to integrate with XMS. There are three possible **integration strategies**:

no integration or limited integration Developers are forced to consult the compiler output to find exact MSIL structure and then put appropriate attributes either at language level or at MSIL level

attribute integration The language recognizes XMS attributes and knowing its own translation schemes puts the attributes in appropriate places inside MSIL

language integration The language syntax is augmented with contract expressions which are compiled as XMS attributes

5 XMS IN PRACTICE

A practical implementation of PCC-oriented certification framework requires three key components: the VCGen that build Verification Conditions for given code modules, a Theorem Prover for Code Producer to build formal proof of a Verification Condition and a Proof Checker for Code Consumer to verify the proof.

The VCGen was exclusively developed for XMS and runs on the .NET platform itself. It reads .NET binaries, scans method bodies and builds Verification Conditions. An example of a session was presented in Section 3.6. Current implementation supports broad range of MSIL instructions, i.a. arithmetical and control flow instructions, instructions for addressing fields and arguments and instructions for calling methods.

There are three possible approaches to theorem proving and proof checking. XMS does not favour any but currently uses the first one.

1. A tactical theorem prover (*Isabelle*, *Coq*) can be used for proof construction and proof validation. Proofs are concise and in many cases can be constructed automatically without any manual guidance. However, the prover must be present at Code Consumer side.

2. Proofs can be encoded in a metalogic (LF [19]). This results in long and detailed proofs but the proof checking procedure is cheap at the Code Consumer side. Metalogic proof checkers are short and thus reliable. Additional techniques can be used to shorten proofs ([12]).

3. A logical interpreter can be used as a proof checker ([13]). Such interpreter uses information about the proof structure provided by the Code Producer but instead of recreating the proof it actually checks if the proof exists at all.

6 SECURE COMPUTATION

One of free benefits of conforming to static verification paradigm with predicates/proofs as certificates is the possibility of using XMS for **Secure Computation**.

Suppose that a party **A** needs expensive computation to be performed on some private data. **A** is unable to perform the computation locally. Suppose that party **B** is able to perform the computation for **A**.

However, **A** does not want its private data to be revealed to **B** and **B** does not want its algorithm to be revealed to **A**.

Using XMS as a certification framework and .NET Web Services as remote computation layer, **A** and **B** can rely on following **XMS Secure Computation Protocol**:

1. **A** and **B** ask a trusted party, **C**, to make a Web Service, **W**, available to both of them

2. **B** publishes its service on **W** together with XMS specification and certificates

3. **A** asks **W** for the specification of **B**'s service, checks if the specification meets his/her requirements and asks **W** to verify that **B**'s service is correct with respect to its specification using XMS Protocol

4. **W** verifies the **B**'s service and sends the verification result to **A**

5. **A** checks the verification status and if it is positive, sends its data to **W** and collects the results

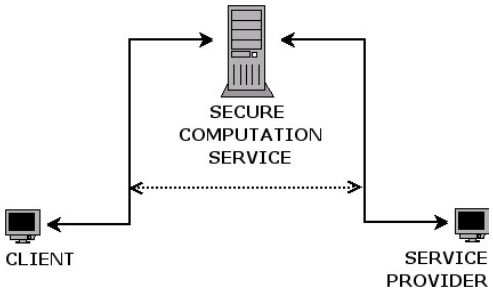

Figure 3: XMS Secure Computation Protocol

7 FUTURE WORK

Formal certificates can rely on other certification paradigms like the Model Carrying Code ([17]) where the certificate takes the form of an abstract model of the code execution and model checking techniques are involved to verify these models. The XMS will ultimately unify various approaches. The combination of PCC and MCC seems especially promising. Three main directions of future XMS development are:

- **support for more MSIL instructions and built-in predicates (Static Verification)**: Currently the static verification does not support all MSIL instructions. It is a short-term implementation goal to support complete MSIL language. For user convenience, some built-in predicates could be supported, such as `ISNULL`.
- **other code instrumentation techniques (Dynamic Verification)**: Although context-bound objects are an easy way to code instrumentation, using .NET Profiler API could make the dynamic verification faster and transparent.
- **better integration with high-level languages**: Current handling of loop invariants require high-level languages to cope with Standard Certificate Specification. This could be restrictive for some high-level languages, for example functional languages with atypical compilation schemes. A long-term goal would be to integrate XMS with such languages using one of proposed integration strategies.
- **other Safety Policies**: Contracts Safety Policy is not the only interesting Safety Policy that can be verified in a XMS manner. Other policies such as Temporal Specifications ([17]) or Non-Interference ([18]) could be adapted to XMS certification scheme.

8 ACKNOWLEDGMENTS

This work is partially supported by MEiN grant 3T11C04230.

REFERENCES

[1] Andrew Bernard and Peter Lee: Enforcing Formal Security Properties, Technical Report *CMU-CS-01-121*, 2001

[2] Andrew Bernard and Peter Lee: Temporal Logic for Proof-Carrying Code, Technical Report *CMU-CS-02-130*, 2002

[3] Andrew D. Gordon and Don Syme: Typing a Multi-Language Intermediate Code, *Microsoft Research*, 2001

[4] Andrew W. Appel: Foundational Proof Carrying Code, *LICS*, 2001

[5] Amy Felty and Andrew W. Appel: Semantic Model of Types and Machine Instructions for Proof-Carrying Code, *POPL*, 2000

[6] Arnd Poetzsch-Heffter and Peter Muller: A Programming Logic for Sequential Java, *ESOP*, 1999

[7] Bertrand Meyer: Applying "Design by Contract", *Computer*, IEEE, Volume 25, Issue 10

[8] Bertrand Meyer: Eiffel: The Language, *Prentice Hall*, 1992

[9] Christopher Colby, Peter Lee and George C. Necula: A Certifying Compiler for Java, 2000

[10] Common Language Infrastructure Specification, *ECMA-335*, 2002

[11] George Ciprian Necula: Compiling with Proofs, *CMU*, 1998

[12] George Ciprian Necula and Peter Lee: Efficient Representation and Validation of Logical Proofs, Technical Report *CMU-CS-97-172*

[13] George C. Necula and S. P. Rahul: Oracle-Based Checking of Untrusted Software, *POPL*, 2001

[14] Greg Morrisett and David Walker: From System F to Typed Assembly Language, 1997

[15] K. Apt and E. Olderog: Verification of Sequential and Concurrent Programs, *Springer-Verlag*, 1997

[16] Mike Gordon: Mechanizing Programming Logics in Higher Order Logic, Springer-Verlag, 1989

[17] R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan and S.A. Smolka: Model Carrying Code: A New Paradigm for Mobile-Code Security,

[18] Riccardo Focardi and Roberto Gorrieri: Classification of Security Properties, *FOSAD*, 2000

[19] Robert Harper, Furio Honsell and Gordon Plotkin: A Framework for Defining Logics, *LICS*, 1987

[20] Mike Barnett, K. Rustan, M. Leino and Wolfram Schulte: The Spec# Programming System: An Overview, CASSIS 2004

[21] Microsoft Corp.: Common Language Infrastructure Specification, ECMA-335 Specification

Aspect-Oriented Programming with Runtime-Generated Subclass Proxies and .NET Dynamic Methods

eva Kühn
eva@complang.tuwien.ac.at

Gerald Fessl
fessl@complang.tuwien.ac.at

Fabian Schmied
fabian@complang.tuwien.ac.at

Institute of Computer Languages, Vienna University of Technology
Argentinierstraße 8
A-1040 Wien, Austria

ABSTRACT

Nine years after its first publication, aspect-oriented programming (AOP) is finding more and more support, but adoption by the industry is still slow. The subclass proxy approach, a new implementation mechansim for .NET-based AOP tools, claims to have the potential of easy adoptability. This paper analyzes subclass proxies as a lightweight infrastructure for AOP, characterizing its properties, advantages, and disadvantages as compared to other implementation techniques. It evaluates technical strengths and weaknesses as well as psychological factors which could influence adoption, and it shows the results of performance benchmarks. In addition, it augments the mechanism with a new way of providing aspects woven at runtime with efficient and safe access to objects' private members.

Keywords

Aspect-oriented programming, code generation, subclass proxies, evaluation.

1. INTRODUCTION

Since 1997, when Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin introduced the term *aspect-oriented programming* (AOP) [KL+97], AOP has found a lot of support in the research community. Successful implementations of the paradigm on the Java platform include AspectJ [LK98], JBoss AOP [Bur04], and Spring AOP [JH+05]. Still, industrial adoption of this new mechanism for software development is naturally slow, and adoption in the field of .NET is hampered by the lack of production-quality AOP tools for this platform.

Recently, projects such as NAspect [Joh05] and XL-AOF [eKS05b] (originally introduced for modelling the concerns of space-based distributed applications in an aspect-oriented fashion [SeK04]) have introduced light-weight implementations of the aspect-oriented programming paradigm based on the .NET platform. Both of them are founded on the same technology, which we call *subclass proxies*. Using subclass proxies as an infrastructure for AOP has some technical advantages over the more classic approaches (i.e. weaving compilers and postcompilers), as well as the important benefit of being easy to adopt by software developers.

In this paper, we perform an analysis of the technology backing both NAspect and XL-AOF. We analyze it on a technical level, describing its implementation, and show its potential as a weaving approach, classify it and characterize it and its properties, advantages, and disadvantages, and we evaluate the concept from an adoptability viewpoint. We also introduce a novel way of allowing aspects to efficiently access private fields and methods of their target objects, which was a privilege of code-weaving approaches until now. By providing a performance analysis, we show that the performance of solutions based on subclass proxies is better than it is often assumed of proxy-based approaches.

The rest of this paper is structured as follows: section 2 describes the technical background of the subclass proxy mechanism and its use for aspect-orientation (note that we use notions such as *aspect*, *advice*, *introduction*, and *join point* as defined by AspectJ [LK98] without further explanation). Section 3 augments the mechanism by introducing an approach for accessing a target object's private secrets from within a subclass proxy-based aspect. Section 4 does an extensive analysis of the concept's potential as a base for AOP and section 5 evaluates it from a performance viewpoint. Section 6 concludes the paper.

.NET Technologies 2006

2. SUBCLASS PROXIES

A *proxy P* is defined to be an object which acts as a placeholder for a target object *T* [GHJV95]. Wherever

T is expected, the proxy can be used instead, transparently extending the target object's behavior or controlling access to it without client code needing to be adapted. *Runtime proxies* are proxies created dynamically at run time, without the programmer having to prepare a dedicated proxy class for every target class.

The Microsoft .NET Common Language Runtime provides a *transparent proxy* [Low03] mechanism whose uses are limited by its functionality and its impact on application design. Design-wise, it requires the proxied object to be derived from the *System.ContextBoundObject* base class. This will not be an option in some cases, in other scenarios it might require unclean changes to the application design, which is contrary to the goals of AOP [KL+97]. Functionally, it is designed for .NET Remoting (i.e. communication between different application domains, processes, or computers) and it cannot extend behavior of an object which is accessed from its own context (including self calls—methods called on the *this* reference), a drawback for realizing a join point model. Introduction can not be implemented using transparent proxies at all. Positive aspects of the mechanism include that proxies are also *created* transparently because the CLR intercepts the *newobj* instruction (*new* in C#, *New* in *Visual Basic .NET*) and returns a proxy instead of the target object. In addition, the CLR automatically corrects the *this* reference within T's methods—it refers to P instead of T, which is important if it is to be passed to other objects.

Looking for an alternative to transparent proxies, it can be noted that the property of substitutability used previously for defining the term "proxy" is similar to the *Liskov Substitution Principle* (LSP) [LW94], which describes the relationship between subtypes. Like a proxy P, which can be substituted for an object T, the LSP states that an object of a subtype can be substituted for one of a supertype. This similarity can be used to implement proxies using the subtyping mechanisms present in .NET: interfaces and inheritance.

To realize a proxy using interface implementation—we call this an *interface proxy* approach—the target object T must implement a set of interfaces I and all client code must access T via these interfaces only. Then, a proxy object P can be created which also implements I and holds a reference to T for delegation. T in the client code can be transparently replaced by P, which plays the role of a proxy. Within T's method implementations, however, the *this* reference refers to T rather than P, which is problematic if the reference is used to access the object: such access will not be registered by the proxy. In addition, like with transparent proxies, the interface proxy approach does not allow self calls to be extended, it is therefore a suboptimal solution as well.

Realizing proxies using inheritance—the *subclass proxy* approach—is different from the aforementioned approaches. Whereas transparent and interface proxies have an object instance P replacing a target object instance T (and delegating), inheritance allows proxy and target to be one and the same object: a class P is derived from the target class T, overriding its methods and delegating to the original implementation. When P is instantiated, one object instance implements both P's and T's functionality. Since subclasses are subtypes in .NET, the LSP applies and instances of P can be used wherever instances of T are expected. Subclass proxies intercept self-calls correctly, the *this* reference is automatically correct, and introduction is possible via interface implementation (see below). Figure 1 compares the unproxied scenario with a simplified drawing of transparent, interface, and subclass proxies.

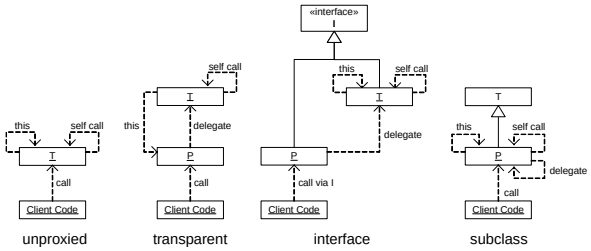

Figure 1. Proxy approach visualization.

In contrast to transparent proxies, both interface and subclass proxy have the disadvantage of needing a class factory [GHJV95] to make object creation transparent to client code. Table 1 summarizes the properties of the different proxy approaches, positive characteristics are shown in boldface.

	Transparent	**Interface**	**Subclass**
Parent class	ContextBoundObject	**arbitrary**	**arbitrary**
Creation	**newobj**	factory	factory
Usage	**direct**	interfaces	**direct**
This reference	**P**	T	**P**
Extend self calls	no	no	**yes**
Introduction	no	**yes**	**yes**

Table 1. Properties of proxy approaches.

2.1 Runtime Subclass Proxies

In the simplest form, subclasses do not implement a runtime proxy approach: the programer needs to write dedicated derived classes for each target type, manually overriding the methods that need to be extended. Using code generation, this can however be generically performed at runtime by a tool or framework. The .NET Base Class Library provides two powerful mechanisms allowing for runtime code generation: the *System.Reflection.Emit* namespace contains low-level classes and methods to dynamically generate .NET assemblies and types, *System.CodeDom* provides base classes for higher-level code generation. In this article, we will concentrate on *System.Reflection.Emit*.

Listing 1 shows how to dynamically generate a subclass of an arbitrary type at runtime using *System.Reflection.Emit*: it asks the current application domain to define a dynamic assembly, naming it "proxies", which in turn is used to define a dynamic module named "proxies" as well; by giving the module a DLL file name (and using the *RunAndSave* flag when creating the assembly), it is possible to save the module to disk after generation in addition to using its types. The dynamic module is used as a factory for the type which is to be created. The type's base class is specified to be *baseType*, the parameter passed to the method (which corresponds to the proxied type *T*), its access attribute is *Public* to make it publicly accessibly from other assemblies, and its name is defined by attaching "___Subclass" to the base type's name. By calling its *CreateType* method, the dynamic type is finished and the corresponding *Type* object (*P*) is returned and can be instantiated using *System.Activator.CreateInstance*.

```
public Type DefineSubclass(Type baseType) {
 AssemblyBuilder a =
   AppDomain.CurrentDomain.DefineDynamicAssembly(new
    AssemblyName("proxies"),
   AssemblyBuilderAccess.RunAndSave);
 ModuleBuilder m = a.DefineDynamicModule("proxies",
   "proxies.dll");
 TypeBuilder subtype = m.DefineType(baseType.Name +
   "___Subclass", TypeAttributes.Public, baseType);
 return subtype.CreateType();
}
```

Listing 1. Creating a subclass at runtime.

2.2 Weaving Based on Subclass Proxies

Aspect-oriented programming is based on two main concepts: join points, i.e. points in the imperative program flow where aspects' advice methods are triggered, and introduction of new members to the aspects' target classes. Both concepts can—to a degree—be implemented with subclass proxies; the method of doing so is described in this section. An analysis on the join point model which is gained from this mechanism is performed later in section 4.

Join Points By overriding the methods of its base class, a proxy class can provide replacement code for them, delegating to the original (base) implementation if necessary, and triggering join points before, after, and instead of (or around) method executions.

With *System.Reflection.Emit*, overriding methods is easily possible by inserting code prior to calling *TypeBuilder.CreateType*. Listing 2 shows how to override all virtual methods of the given base type. It does so by using the .NET *Reflection* mechanism to find all the public and nonpublic instance methods of the base type, checking whether they are virtual, and, if yes, defining a method with the same name and signature. The signature is found by inspecting the parameters of the base method and extracting their types (using an anonymous delegate for brevity); the override's return type is the same as that of the base method.

```
foreach (MethodInfo m in baseType.GetMethods(
   BindingFlags.Public | BindingFlags.NonPublic |
   BindingFlags.Instance)) {
 if (m.IsVirtual) {
   ParameterInfo[] parameters = m.GetParameters();
   Type[] parameterTypes =
      Array.ConvertAll<ParameterInfo,
        Type>(parameters,
      delegate(ParameterInfo parameter)
      { return parameter.ParameterType; });
   MethodBuilder subMethod =
     subtype.DefineMethod(m.Name,
      MethodAttributes.Virtual |
        MethodAttributes.Public,
      m.CallingConvention, m.ReturnType,
        parameterTypes);
   ILGenerator il = subMethod.GetILGenerator();
   il.Emit(OpCodes.Ldarg_0);
   foreach (ParameterInfo parameter in parameters) {
     il.Emit(OpCodes.Ldarg, parameter.Position + 1);
   }
   il.EmitCall(OpCodes.Call, m, null);
   il.Emit(OpCodes.Ret);
 }
}
```

Listing 2. Overriding methods.

The code snippet then defines the override's method body via IL (intermediate language) opcodes. The body loads the object reference (argument 0) and the parameters, calls the base method, and finally returns to the caller. An AOP approach can insert additional code into the body, implementing before, after, and around advice and delegating back to the original method if desired.

Opposed to method join points, construction and creation join points need not be implemented by the proxy itself: they can be triggered by the factory used to create the proxy types and their instances. Property get and set join points are equivalent to method join points, since all properties are backed by respective getter and setter methods. Finalizer join points can be implemented the same way as method join points by overriding the *Finalize* method of the object. Field get and set join points cannot be implemented with subclass proxies. It is up to the AOP implementation of how to bind advice methods to the join points implemented with the subclass proxy mechanism, the most runtime-efficient way being to directly encode calls to advice methods (or even inline these) into the override's method body.

Introduction As opposed to compiler-based AOP approaches, runtime weaving approaches cannot simply introduce new members to a class. While it is easily possible to add these members to a subclass proxy, client code uses the proxy transparently and has no way of accessing the introduced entities with a statically typed programming language. The only form of introduction easily conceivable for runtime approaches is interface introduction: an aspect can add an interface and its implementation to an object, and client code can

cast its object reference to the interface type. Since the proxy implements the interface, the cast succeeds.

Interface introduction can be easily implemented with *Reflection.Emit* by having the dynamically created subclass implement the interface. This is similar to listing 2 and therefore not separately demonstrated here.

3. PRIVILEGED ACCESS TO TARGET OBJECTS' SECRETS

One important property of aspects is that they often require more privileged access to their target object's internals than other objects should have, because they implement cross-cutting concerns which can be tightly coupled to the objects they cut. While a subclass proxy naturally has access to all public and protected (family-accessible) fields and methods of its base class, it has no access to private or assembly-visible members.

.NET provides a Reflection mechanism to work around this: given the necessary rights, every object can reflect over another object's private fields and methods in order to inspect and change the fields' values or invoke the methods. However, Reflection is not optimized for performance: our tests have shown that accessing a field via reflection is around 200 to 700 times slower than direct access, and still 180 times slower than invoking an accessor method would be. Since field access is such a basic operation, this might conceivably slow down an aspect-oriented application, depending on the degree of coupling between aspects and object state.

It would be desirable, therefore, to at least have accessor methods for those private fields required by an aspect. Unfortunately, such a method cannot be added to a subclass, which has no access to private members. As a solution, with .NET 2.0 there is a new mechanism called *Lightweight Code Generation* (LCG), or *Dynamic Methods* [Mic06]. It allows methods to be generated at runtime which can be attached to any existing type, allowing access to all its private data. Access to the method is provided via a delegate, allowing flexible invocation which is still 15 to 20 times faster than reflection-based field access. The diagram in figure 2 shows a performance comparison of the different ways of accessing fields (measured on an Athlon XP1800+ with 512 MB RAM and .NET framework v2.0.50727).

Field Access Framework For an AOP approach based on subclass proxies, we suggest a field access infrastructure which consists of a set of generic delegate types *Setter* and *Getter* as strongly typed wrappers for the accessor methods, an accessor method generator *MethodGenerator* which generates the accessor methods using LCG, and a wrapper structure for fields, which simply initiates the accessor method generation

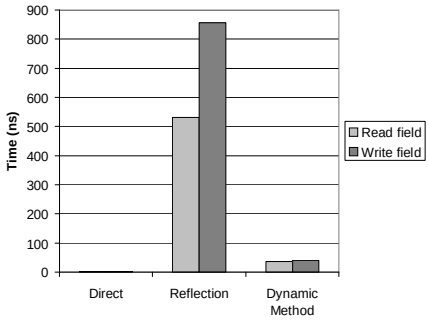

Figure 2. Performance of access methods.

when being constructed and provides a *Value* property delegating to the accessor methods for convenient use.

Listing 3 shows the source code for these infrastructure entities. The method body constructed by *CreateSetter* simply loads the given target object (argument 0) followed by the value (argument 1), which is then stored in the given field before returning. The method body constructed by *CreateGetter* first loads the target, then loads the field value, and then returns, leaving the field value on the evaluation stack, returning it to the caller. Because the created dynamic methods are associated with the target type (ClassType), they can safely access even private fields using the *ldfld* and *stfld* opcodes.

```
delegate FieldType Getter<ClassType, FieldType>(
   ClassType target);
delegate void Setter<ClassType, FieldType>(
   ClassType target, FieldType value);

class MethodGenerator {
 public static Setter<ClassType, FieldType>
   CreateSetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedSetter",
       typeof(void),
     new Type[] { typeof(ClassType),
       typeof(FieldType) },
     typeof(ClassType));
  ILGenerator ilGenerator =
    newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldarg_1);
  ilGenerator.Emit(OpCodes.Stfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Setter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Setter<ClassType, FieldType>));
 }

 public static Getter<ClassType, FieldType>
   CreateGetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedGetter",
     typeof(FieldType), new Type[] {
       typeof(ClassType) },
     typeof(ClassType));
  ILGenerator ilGenerator =
    newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Getter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Getter<ClassType, FieldType>));
 }
```

 ISBN 80-86943-13-5

```
}

struct Field<ClassType, FieldType> {
 public readonly FieldInfo FieldInfo;
 public readonly ClassType Target;
 public readonly Getter<ClassType, FieldType>
   Getter;
 public readonly Setter<ClassType, FieldType>
   Setter;

 public Field(ClassType target, FieldInfo
   fieldInfo) {
  this.FieldInfo = fieldInfo;
  this.Target = target;
  this.Getter = MethodGenerator.CreateGetter
     <ClassType, FieldType>(fieldInfo);
  this.Setter = MethodGenerator.CreateSetter
     <ClassType, FieldType>(fieldInfo);
 }

 public FieldType Value {
  get { return Getter(Target); }
  set { Setter(Target, value); }
 }
}
```

Listing 3. Infrastructure for efficient field access.

4. CONCEPTUAL ANALYSIS

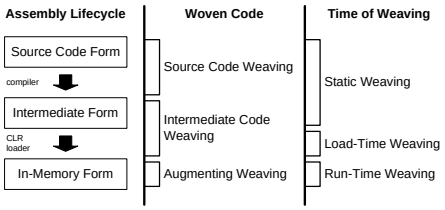

Figure 3. Classification of weaving mechanisms.

Figure 3 shows two orthogonal classifications of weaving mechanisms with regard to an assembly's lifecycle. On the one hand, weaving is classified based on the kind of woven code: *source code weaving* manipulates an assembly's source code to inject aspect or glue code[1]; it can be performed either by a code transformation tool or by a dedicated compiler (plugin). *Intermediate code weaving* manipulates intermediate code (IL and metadata [ECM05]) to inject aspect or glue code; this is usually done by a post-compiler or custom class loader. *Augmenting weaving* does not manipulate existing code, but instead augments it with new glue code, connecting it to aspect code; this can be performed by frameworks rather than tools. Subclass proxy-based approaches are augmenting mechanisms, while traditional approaches like *AspectJ* are source code or intermediate code weaving mechanisms or mixes thereof.

On the other hand, weaving can be classified based on when it is performed: *AspectJ*, for example, has traditionally been a static weaver (including load-time weaving in recent versions) [tAT05]. Contrarily, subclass proxy approaches are real runtime weaving approaches, with the main weaving done at object instantiation time, when the proxy type is created.

[1] *Glue code* is code which "glues" an aspect to its target objects.

With regard to the kind of code being woven, we characterize based on the following properties, displayed in table 2 with advantageous properties in boldface:

Invasiveness is a measure for the degree of manipulation the weaving approach performs on user-written code. Source code weaving approaches compiling a dedicated aspect language have low invasiveness, augmenting approaches only extend and also have low invasiveness. Other approaches change the structure of user-written code and are thus highly invasive.

Debuggability denotes how much effort is needed to make the woven program debuggable with standard mechanisms (e.g. Microsoft Visual Studio). This is easy with proxy-based approaches, because the original debug information remains valid after weaving. Source code weaving also results in correct debug information. With intermediate code weaving, debuggability involves manipulating a debugger-specific file format. This is not portable and often hard: for example, the undocumented *Program Database* file format used by Visual Studio cannot be easily manipulated.

Join point model denotes the join point kinds an approach can provide. Source code weaving makes no restrictions whatsoever to the join point model. With intermediate code weaving, the only restrictions are those posed by IL and metadata (e.g. there are no "for" loops available in IL; to a certain extent this can be overcome by pattern matching as it is also done by decompilers). Augmenting weaving relies on the manipulation mechanisms provided by the CLR, i.e. OOP techniques such as interfaces implementation and method overriding.

Design prerequisites describes prerequisites needed from the perspective of the application designer. With subclass weaving, it is necessary to use a factory to instantiate objects. With source code and intermediate code weaving, there is no such restriction.

Tool prerequisites describes the tools needed for the approach. Source code weaving needs a precompiler or real compiler, intermediate code weaving needs a post-compiler or class loader, augmenting weaving can be done by a framework or library.

Implementation effort is a measure for the effort needed to create a tool based on the approach and keep it up to date with platform changes. Source code weaving requires the most effort by an implementer: it needs at least a source code parser and source code emitter. If the tool is a compiler, complexity is even worse. With intermediate weaving, an IL and metadata parser and emitter are needed, although IL is typically simpler to weave than source code. Augmenting weaving only requires a very simple framework.

Compatibility is a measure for the compatibility of the approach with third-party compilers, frameworks, or

metaprogramming tools. With compiler-based source code weaving tools, third-party compilers cannot be used. Intermediate code and augmenting weaving tools pose no compatibility problems and can usually easily be combined with any compiler, framework, or tool.

Language support denotes the number of supported programming languages. While intermediate code and augmenting weaving strategies can handle all programming languages targeting .NET, a source code weaving tool can only target a single programming language. Since one of .NET's goals is to be a multilanguage environment [ECM05], this is an important restriction which might exclude a high number of potential users (much more important than on the Java platform).

Performance is a measure for the runtime efficiency of the approach. With source code weaving, performance is optimal, all compiler optimizations and JIT optimizations can be performed. With IL code weaving, compiler optimizations should be disabled in order to retain a powerful join point model (e.g. target methods must not be inlined by the compiler), but JIT optimizations can be performed without any restriction. With augmenting weaving, some optimizations are disabled by the use of certain OOP features (like virtual method calls), but most JIT optimizations are available.

	Source	Intermediate	Augmenting
Invasiveness	**low** to high	high	**low**
Debuggability	**no-effort**	hard	**no-effort**
Join point model	**arbitrary**	**IL and metadata**	OOP
Design prerequ.	**none**	**none**	factory
Tool prerequ.	compiler	postcompiler	framework
Impl. effort	very high	high	**low**
Compatibility	low	**high**	**high**
Language support	one	**all**	**all**
Performance	**optimal**	**good**	medium

Table 2. Weaving approaches by code form.

With regard to the time of weaving, we characterize the approaches as follows, summarized in table 3:

Changeability denotes how much effort is needed to add or remove an aspect to or from the application. With static weaving, recompilation or reinstrumentation of the assembly is needed, the application has to be restarted and redeployed. With load-time weaving, the application domain needs to be reloaded, often requiring a restart. With runtime weaving, changes can be applied immediately to objects created after the change.

Deactivating aspects is equally possible in all three weaving variants and requires some sort of join point manager which is asked before a join point is triggered.

Error detection refers to the point of time when weaving configuration errors are detected. With static weaving, this is before application deployment, whereas it is after deployment with the other two approaches.

Testability is inversely proportional to the effort needed to test an object in scenarios with different (or no) aspects attached to it. This follows directly from the changeability: static and load-time weaving require much effort, whereas runtime weaving does not.

	Static	Load-Time	Runtime
Changeability	recompilation	reload	**immediately**
Deactivating aspects	**immediately**	**immediately**	**immediately**
Error detection	**before depl.**	after depl.	after depl.
Testability	low	low	**high**

Table 3. Weaving approaches by time of weaving.

4.1 Join Point Model

From an AOP perspective, a number of join points can be implemented using subclass proxies, whereas others can't. Table 4 characterizes the join point model realizable with the approach. Using a source code weaving tool, all the join points shown could be realized.

Join Point Type	Before	Instead of	After
Object creation	**yes**	**yes**	**yes**
Constructor execution	**yes**	no	**yes**
Class construction	no	no	no
Object finalization	**yes**	**yes**	**yes**
Method execution	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Method call	no	no	no
Property get	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Property set	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Field get	no	no	no
Field set	no	no	no
Exception thrown	no	no	no
Exception caught	no	no	no
Exception escaping	**yes** (virtual)	**yes** (virtual)	-
Construct (for, if, ...)	no	no	no

Table 4. Join point model with subclass proxies.

While this join point model is definitely restricted when compared to that of a source code weaving tool, we believe that this is not a problem in most AOP scenarios. When an application is designed from scratch in an aspect-oriented way, all join points are known in advance, before any of the classes or aspects is to be implemented. With a subclass proxy approach, the design would naturally evolve around the join point kinds being available, ignoring those which can't be used. In most cases, however, small design changes can work around the missing join point types.

For example, because field access join points cannot be realized using subclass proxies, a design guideline could be created to access fields via accessor methods (or properties) only, which is a common guideline with OOP already. Those methods whose execution is needed as a join point would be defined to be virtual. The only join points which can't be worked around are: instead-of constructor execution, class construction, exceptions thrown and caught in the same method, and join points at a statement-level granularity. In addition, subclass proxies cannot advise non-virtual methods or distinguish between method call and execution.

4.2 Psychological Factors

Adoption of AOP is hindered by many factors, which are remedied to a great extent by the use of an approach based on subclass proxies:

AOP as an invasive mechanism: AOP is often regarded with distrust, because its tools weave code together as a black box. Developers can't see what happens when aspects and objects are mangled together, concerns about reliability and debuggability (if an error occurs in mangled code, will it be retraceable to the original source code?) as well as the question of unpredictable execution paths in woven code arise. In contrast, subclass proxies are built on established object-oriented concepts such as method overriding and interface implementation. These are well-known, don't introduce reliability or debuggability problems, and developers can comprehend what happens at runtime.

Adaption to new tools: Aspect-oriented tools often replace the tools (e.g. compilers) developers are used to instead of augmenting them. With all approaches, developers need to adapt to new tools with new error messages, longer or different update cycles, and sometimes incompatibilities with the original tools. Since subclass proxies can be implemented as a framework or class library, there is no need to switch tools with such an approach—developers can continue using their familiar environment and still obtain the benefits of AOP.

Unfinished tools: AOP tools usually need a lot of work, this is the cause of the lack of production quality .NET-based AOP tools. However, since subclass proxies are much simpler to implement than code weaving tools, the probability of reaching production status is much higher with this approach.

AOP based on subclass proxies has high adoption potential. With the described prerequisites, users should be easily convincable of the new technology.

5. PERFORMANCE EVALUATION

With proxy-based approaches, aspect code is not directly inserted into the target code; object-oriented mechanisms are used instead. This is often regarded as a performance disadvantage of such approaches. On the .NET platform, however, most optimizations are not done by a language compiler inlining code, but by the JIT compiler's optimizer at runtime. There are some restrictions to JIT optimization with subclass proxies, because virtual method calls to the target object are always performed through the proxy and can't be replaced by ordinary calls, but these apply just as well when the application makes use of the object-oriented mechanisms itself. Most JIT optimizations should not be affected adversely by the use of subclass proxies.

In this section, we will take a look at two implementations of the subclass proxy mechanism—NAspect and DynamicProxy [Ver04] (which XL-AOF is based on)—and analyze object construction time and method call time, since these represent the main points during program flow where a proxy-based mechanism performs differently from a mechanism based on code weaving.

5.1 Object Creation

The first time an object is created from a target type, the proxy-creating factory must construct the new proxy subclass. This is a lengthy operation, our measurings have shown this to take up to 37ms (DynamicProxy) and 12ms (NAspect), as opposed to the few nanoseconds an ordinary new operation (usually) needs. Seen as isolated numbers, this is a tremendous slowdown.

However, analysis of cross-cutting concerns in space-based computing [eKS05a] reveals that common scenarios only have few types being aspectized at the same time, with a higher number of instances created from those. In such scenarios, the generated proxy subclasses can and should be cached, making an instantiation consist of one hashtable lookup plus one call to the type's constructor (either via Reflection or, optimized, via a delegate), which takes a few hundred microseconds at most in our measurements. In the use cases we studied, this makes instantiation time of proxied objects not a problem. On the other hand, if it is vital that proxied objects of many different types are created with rigid performance requirements (a few nanoseconds per instantiation), pure proxying might not be the mechanism of choice, although pooling and flyweight techniques [GHJV95] can improve on that.

Regarding memory usage, an AOP tool based on subclass proxies should use as few dynamic assemblies and modules as possible. Our tests have shown this to scale much better than having one assembly per proxied type. Caching of the generated proxy types will also improve memory footprint. A user should be aware that the only way to remove the generated proxy types from memory is by unloading their application domain (of course, their *instances* are garbage collected as usual), although again this will not be an issue in scenarios with a reasonable number of aspectized types.

5.2 Method Invocation

Method join point performance is more important than object creation performance because the frequency of method calls as compared to object instantiations is typically very high. With subclass proxies, method join points are implemented via method overrides. A method join point of an optimal proxy is therefore no different from a virtual method call (a few nanoseconds) plus one non-virtual base call if delegation to the original code is needed (a few nanoseconds as well). This optimal approach however requires injection of advice code into the subclass proxy, which is not trivial to implement. Current implementations therefore

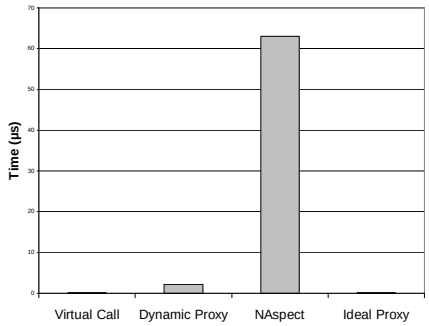

Figure 4. Method call benchmarks.

choose not to directly invoke the base method from within the override. Instead, they encapsulate the base call and hand it to an interceptor provided by the aspect, which may then choose to invoke the method or not. For this encapsulation, DynamicProxy constructs a delegate, whereas NAspect relies on Reflection. Both approaches are not ideal what regards method interception performance, although delegates are an order of magnitude faster than Reflection.

Figure 4 shows a method call benchmark done with an AMD XP1800+ system. The values for ordinary virtual call, DynamicProxy, and NAspect are measured, the value for the ideal proxy is calculated—an implementation can achieve this performance if call times are of much importance. We measured the call and return time of empty methods (with the proxies delegating to the original empty methods); in real scenarios, these values have to be seen in relation to concrete method execution time. For example, our tests have shown that with an average method whose body needs several microseconds for execution, the measured call times are not that significant.

To summarize, while current implementations show medium to significant method call slowdowns, an ideal subclass proxy approach can lead to call times in the range of nanoseconds, not much higher than ordinary method calls. Even the call times of current implementations are less significant if the called methods have nontrivial bodies.

6. CONCLUSION

In this paper, we have motivated, described, and analyzed the subclass proxy mechanism as an implementation infrastructure for aspect-oriented programming. We compared the different proxy mechanisms available on the .NET platform, identifying the subclass proxy mechanism as the most powerful of these. Classifying the weaving approach implementable with subclass proxies, we have shown the disadvantages of the model, such as a more constrained join point model and design restrictions, but have also identified technical advantages over classical implementation mechanisms, such as easy debuggability and runtime weaving capabilities.

Performance benchmarks have shown current subclass proxy implementations to be of medium performance; however the proxy concept could be improved in this regard if necessary in order to achieve call times not much different from ordinary virtual calls.

Analyzing the psychological properties of subclass proxies, we have identified a high potential of the non-invasive mechanism which requires no dedicated compiler tools—we ourselves have successfully used the approach for developing space-based distributed applications [SeK04]. Such light-weight implementations could finally lead to industrial acceptance of aspect-oriented programming.

REFERENCES

[Bur04] Bill Burke. *JBoss Aspect-Oriented Programming (AOP)*, 2004. Available from: `http://www.jboss.org/products/aop`.

[ECM05] ECMA International. Standard ECMA-335 – common language infrastructure (CLI), 3rd edition. Technical report, Ecma International, 2005. Available from: `http://www.ecma-international.org/publications/standards/ecma-335.htm`.

[eKS05a] eva Kühn and Fabian Schmied. Attributes &Co – collaborative applications with declarative shared objects. In *Proceedings WWW/Internet 2005 (Lisbon, Portugal, October 19–22 2005)*, pages 427–434, 2005.

[eKS05b] eva Kühn and Fabian Schmied. XL-AOF – lightweight aspects for space-based computing. In *Proceedings Workshop on Aspect-Oriented Middleware Develoipment (AOMD, Grenoble, France, November 28 2005), Article No. 2*, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, 1995.

[JH+05] Rod Johnson, Juergen Hoeller, et al. *Spring - Java/J2EE Application Framework*, 2005. Available from: `http://static.springframework.org/spring/docs/1.2.x/spring-reference.pdf`.

[Joh05] Roger Johansson. *NAspect AOP engine*, 2005. Available from: `http://blogs.wdevs.com/phirephly/archive/2005/11/23/11308.aspx`.

[KL+97] Gregor Kiczales, John Lamping, et al. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP 98)*, 1998.

[Low03] Juval Lowy. Decouple components by injecting custom services into your object's interception chain. *MSDN Magazine*, March 2003, 2003.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[Mic06] Microsoft Corporation. *DynamicMethod Class*, 2006. Available from: `http://msdn2.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx`.

[SeK04] Fabian Schmied and eva Kühn. Distributed peer-to-peer application development with declarative and aspect-oriented techniques. In *Conference Proceedings of International Symposium on Leveraging Applications of Formal Methods (ISoLA, Paphos, Cyprus, October 30 – November 2 2004)*, pages 150–157, 2004.

[tAT05] the AspectJ Team. *The AspectJ 5 Development Kit Developer's Notebook*, 2005. Available from: `http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[Ver04] Hamilton Verissimo. *Castle's DynamicProxy for .NET*, 2004. Available from: `http://www.codeproject.com/csharp/hamiltondynamicproxy.asp`.

Superinstructions and Replication in the Cacao JVM interpreter

M. Anton Ertl * Christian Thalinger Andreas Krall
TU Wien

Abstract

Dynamic superinstructions and replication can provide large speedups over plain interpretation. In a JVM implementation we have to overcome two problems to realize the full potential of these optimizations: the conflict between superinstructions and the quickening optimization; and the non-relocatability of JVM instructions that can throw exceptions. In this paper, we present solutions for these problems. We also present empirical results: We see speedups of up to a factor of 4 on SpecJVM98 benchmarks from superinstructions with all these problems solved. The contribution of making potentially throwing JVM instructions relocatable is up to a factor of 2. A simple way of dealing with quickening instructions is good enough, if superinstructions are generated in JIT style. Replication has little effect on performance.

1. Introduction

Virtual machine interpreters are a popular programming language implementation technique, because they combine portability, ease of implementation, and fast compilation. E.g., while the Mono implementation of .NET has JIT compilers for seven architectures, it also has an interpreter in order to support other architectures (e.g., HP-PA and Alpha). Mixed-mode systems (such as Sun's HotSpot JVM) employ an interpreter at the start to avoid the overhead of compilation, and use the JIT only on frequently-executed code.

The main disadvantage of interpreters is their run-time speed. There are a number of optimizations that reduce this disadvantage. In this paper we look at dynamic superinstructions (see Section 2.1) and replication (see Section 2.2), in the context of the Cacao JVM interpreter.

While these optimizations are not new, they pose some interesting implementation problems in the context of a JVM implementation, and their effectiveness might differ from that measured in other contexts. The main contributions of this paper are:

- We present a new way of combining dynamic superinstructions with the quickening optimization (Section 3).
- We show how to avoid non-relocatability for VM instruction implementations that may throw exceptions (Section 4).
- We present empirical results for various variants of dynamic superinstructions and replication combined with different approaches to quickening and to throwing JVM instructions (Section 5). This shows which of these issues are important and which ones are not.

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

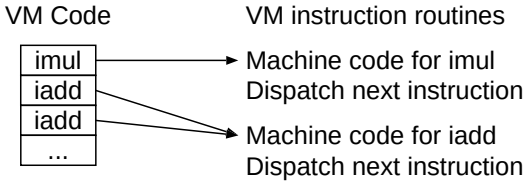

Figure 1. Threaded-code representation of VM code

2. Background

This section explains some of the previous work on which the work in this paper is built.

2.1 Dynamic Superinstructions

This section gives a simplified overview of dynamic superinstructions [RS96, PR98, EG03].

Normally, the implementation of a virtual machine (VM) instruction ends with the dispatch code that executes the next instruction. A particularly efficient representation of VM code is threaded code [Bel73], where each VM instruction is represented by the address of the real-machine code for executing the instruction (Fig. 1); the dispatch code then consists just of fetching this address and jumping there.

A VM superinstruction is a VM instruction that performs the work of a sequence of simple VM instructions. By replacing the simple VM instructions with the superinstruction, the number of dispatches can be reduced and the branch prediction accuracy of the remaining dispatches can be improved [EG03].

One approach for implementing superinstructions is dynamic superinstructions: Whenever the front end[1] of the interpretive system compiles a VM instruction, it copies the real-machine code for the instruction from a template to the end of the current dynamic superinstruction; if the VM instruction is a VM branch, it also copies the dispatch code, ending the superinstruction; the VM branch has to perform a dispatch in order to perform its control flow, otherwise it would just fall through to the code for the next VM instruction. As a result, the real-machine code for the dynamic superinstruction is the concatenation of the real-machine code of the component VM instructions (see Fig. 2).

In addition to the machine code, the front end also produces threaded code; the VM instructions are represented by pointers into the machine code of the superinstruction.

During execution of the code in Fig. 2, a branch to the `iload b` performs a dispatch through the first VM instruction slot, resulting in the execution of the dynamic superinstruc-

[1] More generally, the subsystem that generates threaded code, e.g., the loader or, in the case of the Cacao interpreter, the JIT compiler.

 ISBN 80-86943-13-5

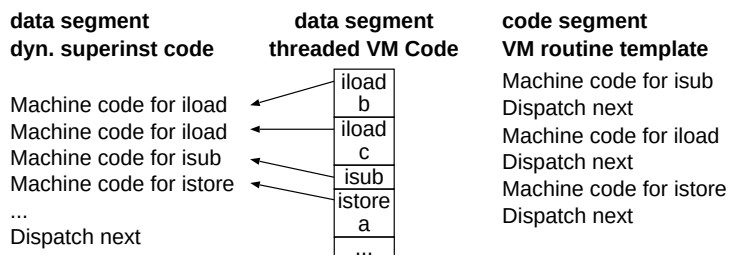

Figure 2. A dynamic superinstruction for a simple JVM sequence

tion code starting at the first instance of the machine code for `iload`, and continues the execution of the dynamic superinstruction until it finally executes another dispatch through another VM instruction slot to another dynamic superinstruction.

As a result, most of the dispatches are eliminated, and the rest have a much better prediction accuracy on CPUs with branch target buffers, thus eliminating most of the dispatch overhead. In particular, there is no dispatch overhead in straight-line code.[2]

There is one catch, however: Not all VM instruction implementations work correctly when executed in another place. E.g., if a piece of code contains a relative address for something outside the piece of code (e.g., a function call on the IA32 architecture), that relative address would refer to the wrong address after copying; therefore this piece of code is not relocatable for our purposes.[3] The way to deal with this problem is to end the current dynamic superinstruction before the non-relocatable VM instruction, let the VM instruction slot for the non-relocatable VM instruction point to its original template code (which works correctly in this place), and start the next superinstruction only afterwards.

Dynamic superinstructions can provide a large speedup at a relatively modest implementation cost (a few days even with the additional issues discussed in this paper). It does require a bit of platform-specific code for flushing the instruction cache (usually one line of code per platform), but if this code is not available for a platform, one can fall back to the plain threaded-code interpreter on that platform.

2.2 Replication

As described above, two equal sequences of VM instructions result in two copies of the real-machine code for the superinstruction (replication [EG03]).

An alternative is to check, after generating a superinstruction, whether its real-machine code is the same as that for a superinstruction that was created earlier[4]; if so, the threaded-code pointers can be directed to the first instance of the real-machine code, and the new instance could be freed (non-replication).

Replication is good for indirect branch prediction accuracy on CPUs with branch target buffers (BTBs) and is easier to implement, whereas non-replication reduces the real-machine code size significantly and can reduce the I-cache misses.

ACONST	INVOKESPECIAL
ARRAYCHECKCAST	INVOKESTATIC
CHECKCAST	INVOKEVIRTUAL
GETFIELD_CELL	MULTIANEWARRAY
GETFIELD_INT	NATIVECALL
GETFIELD_LONG	PUTFIELD_CELL
GETSTATIC_CELL	PUTFIELD_INT
GETSTATIC_INT	PUTFIELD_LONG
GETSTATIC_LONG	PUTSTATIC_CELL
INSTANCEOF	PUTSTATIC_INT
INVOKEINTERFACE	PUTSTATIC_LONG

Figure 3. Instructions in the (JVM-derived) Cacao interpreter VM that reference the constant pool

2.3 Cacao interpreter

For the present work, we revitalized the Cacao interpreter [EGKP02] and adapted it to the changes in the Cacao system since the original implementation (in particular, quickening, and OS-supported threads).

The most unusal thing about the Cacao interpreter is that it does not interpret the JVM byte code directly; instead, a kind of JIT compiler (actually a stripped-down variant of the normal Cacao JIT compiler) translates the byte code into threaded code for a very JVM-like VM, which is then interpreted. The advantage of this approach is that the interpreter can use the fast threaded-code dispatch, and the immediate arguments of the VM instructions can be accessed much faster, because they are properly aligned and byte-ordered for the architecture at hand. Moreover, this makes it easier to implement dynamic superinstructions and enables some minor optimizations.

The Cacao interpreter is implemented using Vmgen [EGKP02], which supports a number of optimizations (e.g., keeping the top-of-stack in a register), making our baseline interpreter already quite fast.

3. Quickening

This section discusses one of the problems of the JVM and .NET when implementing dynamic superinstructions.

3.1 The problem

A number of JVM instructions (see Fig. 3) refer to the constant pool, and through it to components of (possibly) other

[2] Some people already consider this to be a simple form of JIT compilation. In this paper we refer to it as an interpreter technique, for the following reasons: 1) It can be added with relatively little effort and very portably (with fall-back to plain threaded code if necessary) to an existing threaded-code interpreter; 2) The executed machine code still accesses the VM code for immediate arguments and for control flow.

[3] Why do we not support a more sophisticated way of relocating code that does not have this problem? Because that relocation method would be architecture-specific, and thus we would lose the portability advantage of interpreters; it would also make the implementation significantly more complex, reducing the simplicity advantage.

[4] Of course, instead of checking the real-machine code afterwards, one could also check the virtual-machine code beforehand, but that is an implementation detail.

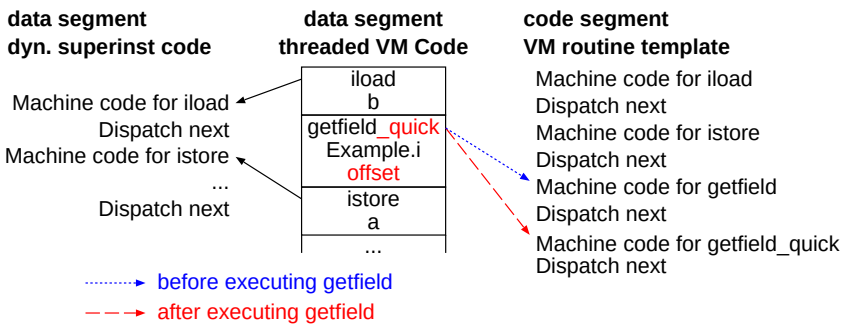

Figure 4. Simple solution: Exclude slow and quick instructions from dynamic superinstructions.

classes. A class should be loaded and must be initialized exactly when the first instruction refering to it is executed.

Performing all the overhead of checking whether the class is already loaded and initialized, and resolving the class and component information into the actual information needed (an offset in the case of `getfield`) on every execution is very expensive: in experiments with an old version of Kaffe[5] we found that optimizing this overhead away gives a speedup on the SpecJVM98 benchmarks by about a factor of 3.

The original Java interpreter optimizes such *slow instructions* by rewriting them and their immediate operand(s) in the VM code into *quick instructions* when they are executed the first time [LY97, Chapter 9]. This optimization is called *quickening*.

The immediate operand of the quick instruction is the result of resolving the operand of the slow instruction. E.g., for the slow `getfield` instruction the immediate operand is a constant-pool reference to the field of a class, whereas the immediate operand of `getfield_quick` is the offset of the field. In our examples (e.g., Fig. 4) as well as in our implementation, we have separate slots for these two operands; in the examples, this makes it easier to show what is happening; in the implementation, this reduces the need for locking [GH03].

This approach does not work with dynamic superinstructions: In general, rewriting the VM code is not enough; we would also have to rewrite or patch the real-machine code generated for the superinstruction; and the difficulties there are that the real-machine code of the slow and the quick instruction usually have a different length; moreover, the slow instruction and its quick equivalent might not be both relocatable (usually, the slow instruction is not relocatable, and the quick instruction is).

3.2 A simple solution

A simple solution is to exclude slow instructions from being integrated into dynamic superinstructions (just as it is done for non-relocatable instructions). A preceding dynamic superinstruction would end right before the slow instruction and dispatch to the slow instruction as usual in threaded code. The slow instruction could then rewrite itself into the quick instruction, as in a plain threaded-code interpreter.

The disadvantages of this solution are:

- Usually two additional VM instruction dispatches are performed per execution of the quick instruction that would not be performed if it was integrated in the dynamic superinstruction: One for ending the preceding superinstruction, and one by the quick instruction itself. This hurts mainly CPUs without BTBs (branch target buffers) or similar indirect-branch predictors.
- The quick instruction is not replicated, leading to a low prediction accuracy for the dispatch by the quick instruction on CPUs with BTBs. This disadvantage could be eliminated in, e.g., the following way: When the slow instruction rewrites itself into the quick instruction, it replicates the quick instruction (including its dispatch) and lets the instruction slot point to the new replica. However, this approach will lead to less spatial locality and thus more I-cache misses than the normal arrangement of dynamic superinstructions with replication.
- When applying additional optimizations, such as static superinstructions [EGKP02] or static stack caching [EG04a], the natural approach to take would be to also exclude the to-be-quickened instructions from these optimizations. Everything else would require additional implementation costs similar to more sophisticated approaches for this problem.

These disadvantages lead to significant slowdowns (compared to more sophisticated approaches) when all slow instructions are treated in this way [GH03].

However, the Cacao interpreter translates the JVM bytecode into threaded code using a JIT compiler with method granularity. If the JIT compiler encounters a slow JVM instruction that references a class that has already been loaded and initialized, it translates it into a quick instruction without the intermediate state of a slow threaded-code instruction. These quick instructions can be integrated into dynamic superinstructions without a problem.

So, in the Cacao interpreter, only a subset of the slow instructions from the original JVM code have the problems mentioned above even with this simple solution. If the parts of the code containing this subset are only executed rarely, the performance disadvantage of the simple solution is negligible. Our results (see Section 5) indicate that this is indeed the case.

However, we did not know this from the start, so we also looked into more sophisticated approaches. Moreover, more sophisticated approaches do have their merits in settings (like SableVM) where no JIT translation into threaded code with superinstructions is used: At least our sophisticated approach is simpler to implement than a JIT translator.

3.3 Previous sophisticated solutions

Like the Cacao interpreter, SableVM translates the JVM bytecode into threaded code with dynamic superinstructions. One difference is that SableVM keeps only the instruction slot for the first VM instruction in a superinstruction, whereas the Cacao interpreter keeps all the VM instruction slots around (even

[5] http://www.complang.tuwien.ac.at/java/kaffe-threaded/

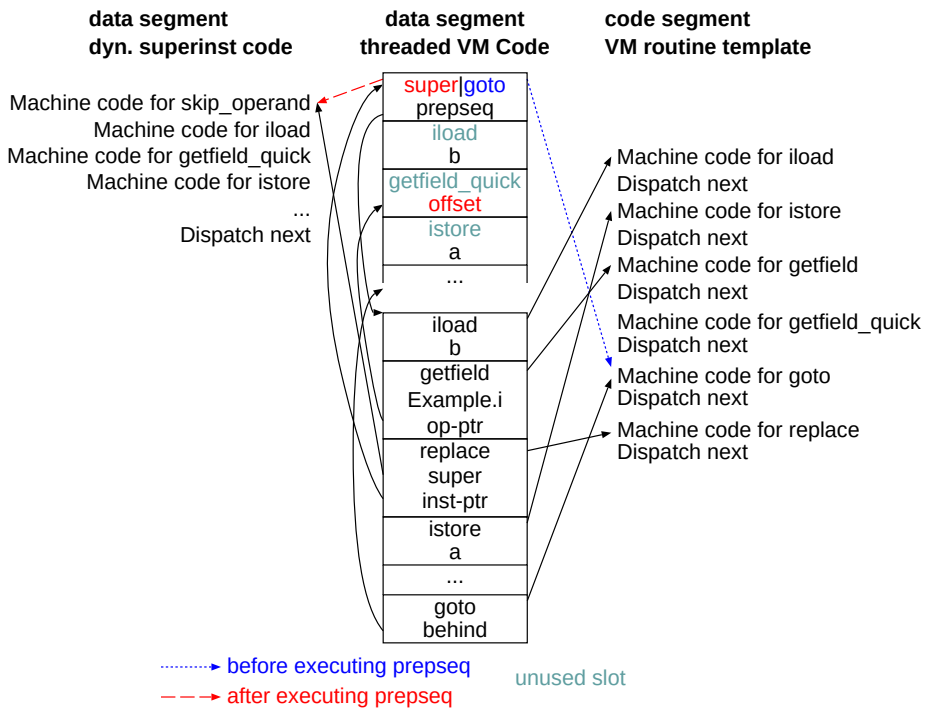

Figure 5. SableVM's preparation sequence for the first execution and the dynamic superinstruction including a quick instruction for subsequent executions

though only the first one is used when the superinstruction is executed); to avoid confusion, we show the same approach in all the examples: all VM instruction slots are kept.

SableVM deals with quickening by creating an out-of-line preparation sequence in VM code (see Fig. 5), as well as the superinstruction (which incorporates the quick versions of any instructions to be quickened instruction). On first execution the VM code jumps to the preparation sequence, which performs the first execution (including a variant of the slow instruction that patches the operand elsewhere), and rewrites the `goto` to the preparation sequence into an invocation of the superinstruction; finally, the preparation sequence jumps to the first (super)instruction behind the VM code covered by the superinstruction. On the next execution, the VM code just executes the superinstruction.

Casey et al. [CEG05, Section 5.4] have also implemented dynamic superinstructions in a JVM interpreter. They treat the slow instruction as non-relocatable, as in the simple solution, but leave space in the real-machine-code area for the (real-machine code of the) corresponding quick instruction; on quickening, they copy the real-machine code for the quick instruction into that space, resulting in a dynamic superinstruction that includes the quick instruction. This solution requires that all VM instruction slots are kept around.

3.4 Our sophisticated solution

Figure 6 shows our approach: When we generate the threaded code for a block, we also generate the real-machine code for the superinstruction; however, if we encounter a slow instruction, we generate the real-machine code for the appropriate quick instruction.

However, if there is a slow instruction in the block, we do not let the threaded code point to the dynamic superinstruction right away. Instead, we first generate conventional threaded code, which does not reference the dynamic superinstruction in any way, and that code contains the slow instructions. We also record what the last slow instruction in the block is, and use this in a table called superstart: the last slow instruction is the lookup key, and it also contains a pointer to the superinstruction real-machine code for the block, and the first threaded-code word in the block.

When a slow instruction is executed, it first performs all the necessary loading and initialization work. Then it looks itself up in the superstart table, and patches the threaded-code word at the block start to point to the real-machine code for the superinstruction.[6] The next time the basic block is executed, it will use the dynamic superinstruction.

We did not define above what we mean by *block*: It is the VM code covered by a dynamic superinstruction. It is essentially the same as a basic block, with one additional boundary condition: If there is a VM instruction with non-relocatable real-machine code, that also terminates the superinstruction (and thus the block); the next superinstruction starts after the non-relocatable VM instruction.

In earlier work [EG03] we let superinstructions continue straight-line across control-flow joins. We cannot do this here; consider the case of a superinstruction consisting of two basic blocks, with each basic block containing one slow VM instruction:

- When the first slow instruction is reached, this is not the last slow instruction in the superinstruction, so we cannot do the patching; if we did, we would get a race condition: another thread could execute the quick instruction implementation in the superinstruction before this thread has performed the necessary class loading and initializations.

[6] We need to patch only the first threaded-code word, because, once we are executing the dynamic superinstruction, the other threaded-code words are not used.

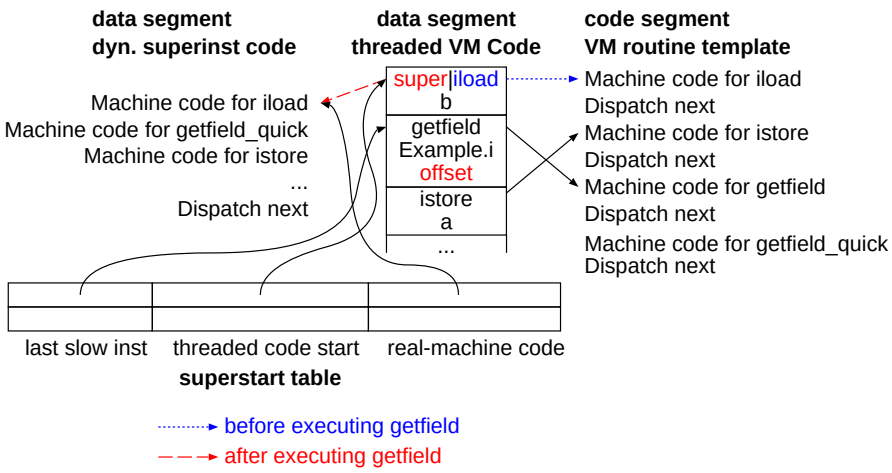

Figure 6. Cacao's sophisticated solution: first execute threaded code; the last slow instruction rewrites the first instruction in the sequence into the superinstruction.

- When the second slow instruction is reached, it does not know if it can patch the start of the first basic block, because it does not know if that basic block and its slow instruction has been executed, and the appropriate initializations done.

As a result, the part of the superinstruction for the first basic block would never be used.

In earlier work we also let superinstructions continue across fall-through edges of conditional branches. We also do not do this here: If there is a slow instruction in the fall-through path, but the branch is always taken, the superinstruction might never be activated.

One could work around these issues, but that would require significant complexity.

Note that the simple solution (Section 3.2) does not have these restrictions and thus can be better than our sophisticated solution (depending on the dynamic frequencies of originally-slow instructions vs. basic block ends and not-taken conditional branches).

Another thing worth noting is that our solution requires that the superinstruction keeps all the VM instruction slots, because the first time the code is executed as plain threaded code. In terms of the SableVM solution, we use the original sequence combined with the entry in the superstart table as preparation sequence. So keeping all the slots leads to a significant simplification here, as well as in other contexts, such as superinstructions across basic block boundaries [EG03] and optimal selection of static superinstructions.

Finally, one of the advantages of our sophisticated approach over the simple solution and over the solution of Casey et al. [CEG05] is that our solution is easier to adapt to situations where dynamic superinstructions are combined with static stack caching and/or static superinstructions: While generating the dynamic superinstruction, we use static stack caching or static superinstructions without having to consider complications from quickening, and the threaded code for the first execution need not use these optimizations.

4. Relocatability and exceptions

Only relocatable real-machine code can be used in dynamic superinstructions (see Section 2.1). In order to be relocatable, a code fragment must not contain relative references to targets outside the code fragment, nor absolute references to targets inside the code fragment.

IALOAD	GETFIELD_CELL
LALOAD	GETFIELD_INT
AALOAD	GETFIELD_LONG
BALOAD	PUTFIELD_CELL
CALOAD	PUTFIELD_INT
SALOAD	PUTFIELD_LONG
IASTORE	INVOKEVIRTUAL
LASTORE	INVOKESPECIAL
BASTORE	INVOKEINTERFACE
CASTORE	ARRAYLENGTH
IDIV	CHECKNULL
IREM	

Figure 7. Instructions in the (JVM-derived) Cacao interpreter VM that can throw exceptions

There are a number of JVM instructions that can throw an exception, but usually don't (see Fig. 7); e.g., `getfield` (and its quick variants) can throw a null pointer exception.

The code for throwing an exception is quite complex, so we don't want to replicate it with frequently occuring instructions like `getfield`. Moreover, it involves a function call, which makes the code non-relocatable on most architectures (it is a relative reference to code outside the fragment).

What we actually would like to do is to keep the throw code common, and jump to it from the various potentially exception-generating VM instructions. Unfortunately, when implemented directly, this usually still makes the exception-generating VM instructions non-relocatable, because the direct jump uses relative addressing on most architectures.

Our way to deal with this is to use an indirect jump instead of the direct jump. Since exceptions are rarely thrown and, when thrown, cost a lot of time anyway, the additional cost of the indirect jump is negligible.

We implement the indirect call by putting the addresses of the throw code in a local variable, and then jumping to it with `goto *`. We have to take some care to confuse the constant propagation[7], otherwise gcc will "optimize" the indirect

[7] We make the local variable appear to be non-constant by having an assignment of another value to it in some code fragment that appears to be reachable.

branch back into a direct branch. An additional problem is that we have to work around the bugs that recent gccs have in this area: PR15242 and PR25285.

Both SableVM [GH03] and Casey et al.'s work [CEG05] solve this problem in a way similar to our's[8], but they do not discuss it in their papers, and do not provide data about the effectiveness of this work.

5. Empirical results

5.1 Setup

The hardware we used in our experiments was a 2.2GHz Athlon 64 X2 4400+, a 2.26GHz Pentium 4, and a 1GHz Pentium III. The main difference between these CPUs for our experiments is in the size of the instruction cache: while the I-cache of the Athlon 64 is relatively large (64KB), it is much smaller in the Pentium III (16KB) and the Pentium 4 (12K microinstructions); so, negative effects of replication should become visible on the latter CPUs first.

All systems were running under Linux 2.6.13 or 2.6.14. We used SpecJVM98 as benchmark; we ran each benchmark three times, and report the median result.

5.2 Superinstructions

We benchmarked a threaded-code Cacao interpreter without any kind of superinstructions (**plain**), and the Cacao interpreter with dynamic superinstructions with all combinations of the following variants:

throw Instructions that can throw an exception cannot (-) or can (+) be integrated in a dynamic superinstruction (Section 4).

simple/soph The approach used for dealing with quickening: simple (Section 3.2) or our sophisticated solution (Section 3.4).

replication Without or with replication (Section 2.2).

We compiled the Cacao interpreter with gcc-2.95. We used GNU Classpath 0.19 as Java library for Cacao.

One thing worth noting is that the performance of the Cacao interpreter is strongly influenced by how many VM registers end up in real-machine registers. In the present case we managed to get the most important VM registers (ip, sp, TOS) in real-machine registers, but with more recent gcc versions, or when compiling the interpreter into a dynamically linkable library, the results are significantly worse. We used the same interpreter executable for all these measurements, with the variants determined by command-line options. This ensures that all the variants use the same register allocation.

Figure 8, 9 show the timing results; for space reasons we do not show the Pentium III results, but they are similar to the Athlon 64 X2 results.

We see that the best variant of dynamic superinstructions provides a huge speedup over plain threaded code, comparable to the effects we saw for Forth [EG03]. The speedup is even bigger on the Pentium 4 (which we did not measure earlier), probably because this CPU has a relatively higher branch misprediction penalty.

Looking at the variations, we see that **throw** has a large performance effect. By contrast, both replication and our sophisticated quickening usually have a small and not consistently positive effect on performance.

Our result for our sophisticated quickening is remarkable because the results for SableVM show a large speedup of sophisticated quickening over simple quickening [GH03]. Our explanation for this is that Cacao converts many instructions (and apparently most of the frequently-executed ones) into quick instructions already during the translation from bytecode (so there is no need to quicken them at run-time and they can be integrated into superinstructions like ordinary instructions), whereas SableVM goes through the slow-instruction stage for all slow instructions in the bytecode.

Another interesting result is that, despite Java's reputation for bloat, replication does not hurt much on any of the benchmarks, not even on the Pentium 4 and III with their small I-caches. So at least the SpecJVM98 benchmarks have good temporal locality. Implementing the non-replication option cost only three hours of work, so it may still be worthwhile (as an option) for CPUs that do not predict indirect branches with BTBs.

5.3 Other systems

Figure 10 shows the performance of various other JVM systems, both interpreters and JIT/mixed-mode systems compared to the Cacao interpreter with dynamic superinstructions.

The first interesting result is that already the **plain** Cacao interpreter (without superinstructions) is quite competetive. Surprisingly, it regularly beats even SableVM (which does use dynamic superinstructions), probably thanks to better register allocation.

The Cacao interpreter with dynamic superinstructions (**+throw soph +repl**) is quite a bit faster, as discussed above.

JIT and mixed-mode systems are generally even faster (except, usually, Kaffe). The most comparable of these is, of course, the Cacao JIT compiler, which provides speedups by up to a factor of 3.3. So, dynamic superinstructions provide performance that is halfway between plain threaded code and a JIT compiler, for much less than half the effort.

6. Related work

The work most closely related to our work is the work on dynamic superinstructions in the JVM in SableVM [GH03] and by Casey et al. [CEG05, Section 5.4]. Both papers discuss the problem of combining quickening with dynamic superinstructions; the sophisticated solutions they present are more complex than our sophisticated solution (for a more detailed discussion, read Section 3.3). One significant difference is that we use a JIT-style translation, which allows us to use quick instructions right from the start in many cases, and this makes the simple approach competetive, whereas SableVM always goes through the slow instructions, and sees a big slowdown from the simple approach. Another difference between our work and the previous ones is that we discuss the issue of the relocatability of instructions that can throw exceptions, and we present results.

Choi et al. [CGHS99] point out the large effect that potential exception-throwing instructions have on a JIT compiler and present some solutions in that context, but do not discuss or solve the problems that are addressed in the present paper.

With static superinstructions the set of superinstructions is fixed at interpreter build time (or earlier). Static superinstructions, and the related, but more complex concepts of supercombinators [Hug82] and superoperators [Pro95, HATvdW99] have been used for a long time in interpreters. This includes an earlier version of the Cacao interpreter [EGKP02]; in that work we did not encounter the conflict between superinstructions and quickening, because that version of Cacao (incorrectly) initialized classes on compiling, not on first execution. So one of the advances of this work over the earlier work is a proper solution for this conflict. The other important differ-

[8] Email communications with Etienne Gagnon and David Gregg.

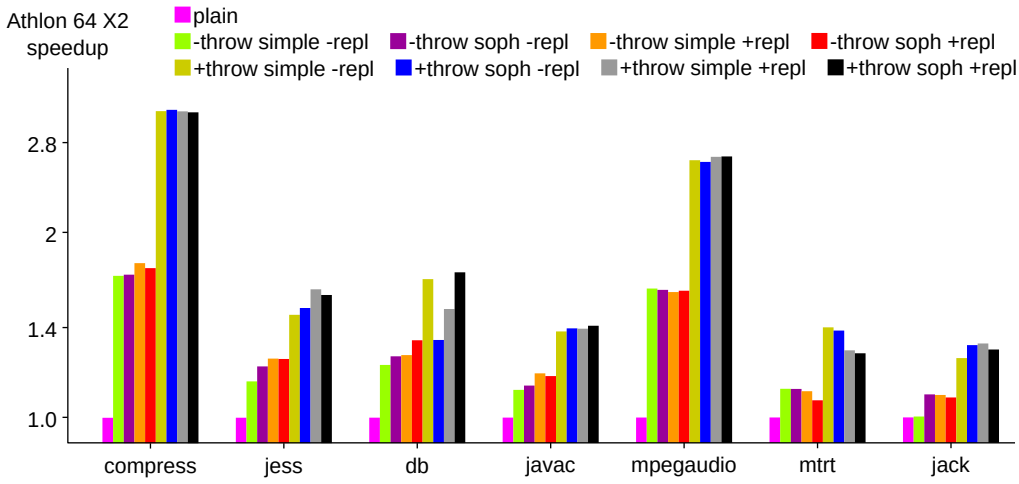

Figure 8. Speedup of dynamic superinstruction variants over *plain* on an Athlon 64 X2 4400+ (log. scale)

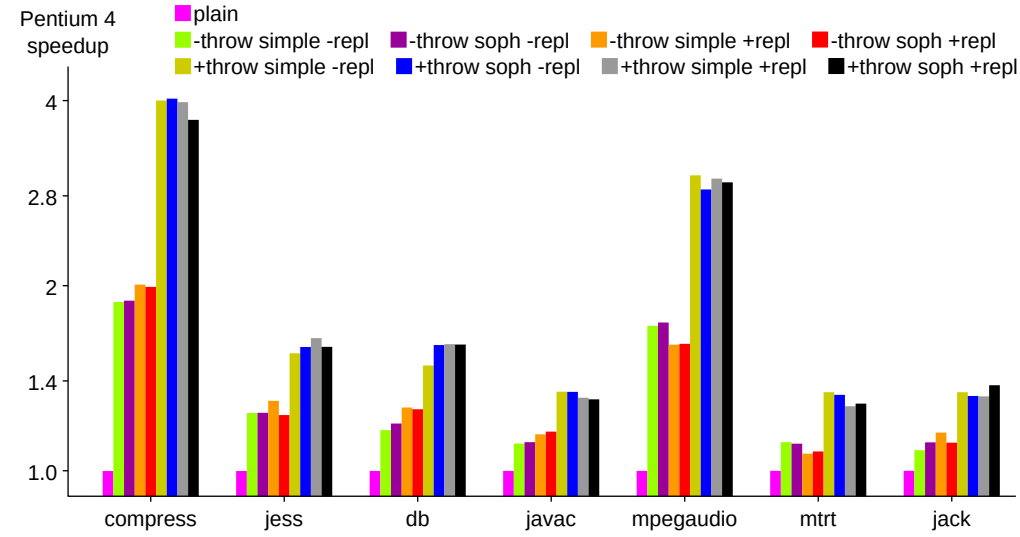

Figure 9. Speedup of dynamic superinstruction variants over *plain* on a 2.26GHz Pentium 4 (log. scale)

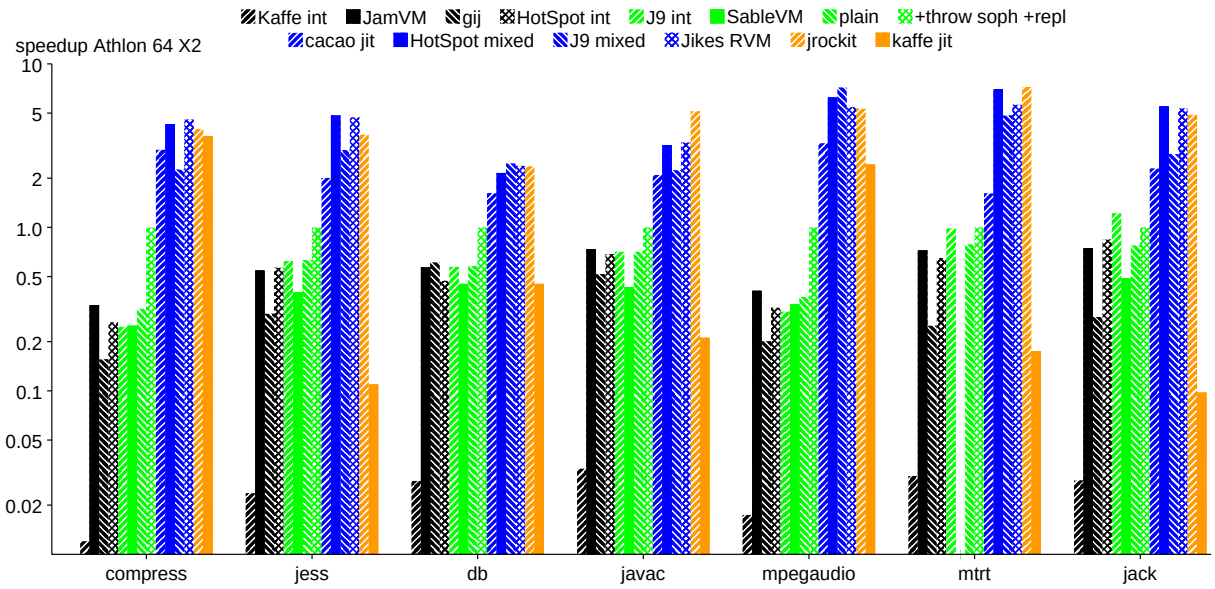

Figure 10. Relative speeds of various JVM systems on an Athlon 64 X2 4400+ (log. scale)

ence between these earlier works and this work is that in this work we look at dynamic superinstructions.

Dynamic superinstructions [RS96, PR98] (also known as selective inlining) are a relatively recent invention. Replication [EG03] was developed to improve BTB indirect branch prediction accuracy, and combines nicely with dynamic superinstructions for improved performance with reduced implementation effort. The present work applies these concepts in the context of the JVM, and solves the problems that arise in this context: combining dynamic superinstructions with the quickening optimization; and ensuring that VM instructions that can throw exceptions can be included in superinstructions.

As a further step after dynamic superinstructions with replication, one can generate code that includes immediate arguments and performs control flow directly instead of through the threaded code, turning the system into a simple native-code compiler. The work based on Forth [EG04b] showed a nice speedup, but the work based on Tcl [VA04] did not show a speedup over the baseline interpreter (without superinstructions or replication) for many application benchmarks, because it led to a large increase in I-cache misses. This problem would certainly also arise in an implementation of dynamic superinstructions with replication (where the resulting code is typically a little bit larger than for the more advanced technique above). Therefore, we were a little worried, how well dynamic superinstructions and especially replication would work for the JVM; we answer these questions in the present work.

7. Conclusion

Applying dynamic superinstructions and replication to the JVM poses two challenges, which we solve in this paper:

- These optimizations conflict with the *quickening* optimization for the first-execution resolution of constant-pool references. A simple approach just excludes slow instructions from dynamic superinstructions. As our empirical results show, this method works well enough in the context of a JIT-style compiler with method granularity, because it usually translates slow instructions to quick instructions already when generating the dynamic superinstruction.

  We also present a more sophisticated approach that is easier to implement than previous sophisticated approaches and is useful if the system does not use JIT-style translation to threaded code.

- Instructions that can throw an exception would normally have non-relocatable real-machine code and could not be included in dynamic superinstructions, leaving a lot of the speedup potential from dynamic superinstructions unused. We solve this problem by converting the direct branches to the throwing code (which are the cause of the non-relocatability) into indirect branches (which are relocatable).

We also present empirical results on a number of platforms: The overall speedup we see is quite large, up to a factor of 4, with a factor of about 2 being more typical. The effect of making instructions that can throw exceptions relocatable is also quite large (up to a factor of 2). Replication has a relatively small effect. A simple approach to quickening combined with a JIT-style translation into threaded code with dynamic superinstructions usually works about as well as a sophisticated approach to quickening.

Acknowledgments

The anonymous reviewers provided comments that helped improve this paper.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[CEG05] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. Submitted to ACM TOPLAS, 2005.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Program Analysis for Software Tools and Engineering (PASTE'99)*, 1999.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[EG04a] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *IVME '04 Proceedings*, pages 7–14, 2004.

[EG04b] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 170–184. Springer, 2003.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[Hug82] R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, first edition edition, 1997.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.