Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Efficient Compilation of .NET Programs for Embedded Systems

Olivier Sallenave[ab] Roland Ducournau[b]

a. Cortus S.A., Montpellier, France
http://www.cortus.com

b. LIRMM — CNRS and Université Montpellier 2, France
http://www.lirmm.fr

Abstract Compiling under the closed-world assumption (CWA) has been shown to be an appropriate way for implementing object-oriented languages such as Java on low-end embedded systems. In this paper, we explore the implications of using whole program optimizations such as Rapid Type Analysis (RTA) and coloring on programs targeting the .Net infrastructure. We extended RTA so that it takes into account .Net specific features such as (i) *array covariance*, a language feature also supported in Java, (ii) *generics*, whose specifications in .Net impacts type analysis and (iii) *delegates*, which encapsulate methods within objects. We also use an *intraprocedural control flow analysis* in addition to RTA . We evaluated the optimizations that we implemented on programs written in C#. Preliminary results show a noticeable reduction of the code size, class hierarchy and polymorphism of the programs we optimize. Array covariance is safe in almost all cases, and some delegate calls can be implemented as direct calls.

Keywords array covariance, closed-world assumption, delegate, late binding, subtype test, type analysis

1 Introduction

High-end embedded systems such as mobile phones have widely adopted object-oriented languages like Java. Object-oriented programming offers code reuse through inheritance and more reliable designs, which improves the productivity in software development. As the complexity of embedded systems is increasing, it has become important to reduce the length of their development cycles. But low-end embedded systems such as SIM cards and sensors are still programmed in C and assembly — the use of object-oriented languages generates an overhead at runtime, which is hard to reconcile with the limited resources of these systems.

We aim at reducing the overhead of .NET for low-end embedded systems. We do not consider high-end embedded systems such as tablet computers or mobile phones, which have a large amount of external memory available. For the systems we are considering, all the code memory is integrated on the chip, therefore the cost of a chip is almost directly proportional to the code space. Also the power consumption is higher if the memory is bigger. As a result, we must focus on both runtime efficiency and memory consumption.

There are several sources of overhead associated with JAVA and .NET — namely dynamic compilation, garbage collection, object mechanisms[1] and the increase in code size due to methods that are never invoked in the libraries. Static compilation resolves the first issue, and we do not consider garbage collection in this paper. To address the last two issues, whole program optimizations such as type analysis and coloring have been commonly used. Type analysis tries to resolve the behavior of object mechanisms at compile-time and identifies some uncallable methods. Coloring can be used to efficiently implement multiple subtyping of interfaces. These techniques require to know the whole program to compile, which means they are not directly compatible with dynamic loading. They require the closed-world assumption (CWA), which ensures that what is known to be true at compile-time will remain true — i.e., the program is entirely known and cannot change at runtime. Most efficient compilation schemes are based on that assumption [DMP09], though current JAVA and .NET runtime systems rely on adaptive compilation in order to support dynamic loading [AFG+04].

We explore the implications of using RTA and coloring to implement .NET on low-end embedded systems. We extended RTA so that it takes into account some .NET specific features:

- *array covariance*, a language feature also supported in JAVA which generates some *subtype tests* at runtime (also called *array store checks*),
- *generics*, whose specifications in .NET allow the programmer to instantiate *open generic types* (`new T`), which brings alive all *generic instances* of `T`,
- *delegates*, which encapsulate methods within objects (only in .NET). Invoking a delegate generally results in one or many *indirect calls*.

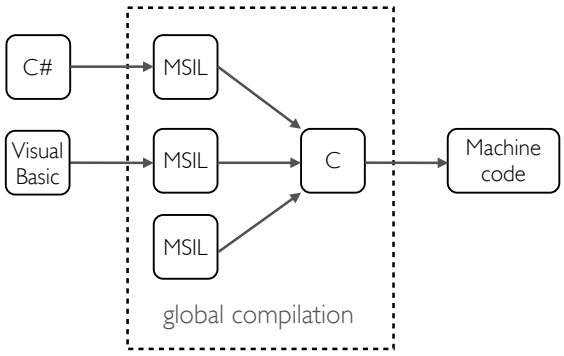
In particular, we target systems based on the Cortus APS3 processor family, which is specifically designed for the embedded world. These 32-bit processors feature a tiny silicon footprint (the same size as an 8-bit Intel 8051), low power consumption, high code density and multi-core configuration [KM10].

This article is structured as follows. Section 2 is an overview of the compiler and runtime. Section 3 presents the type analysis algorithm and the global optimizations we implement. Section 4 explains how we lay down runtime structures. Section 5 presents the results we obtained optimizing a selection of programs that target the .NET platform. The last sections describe related work, conclusion and prospects.

2 Compiler Overview

This section briefly presents our overall implementation of the Common Language Infrastructure [MR04].

[1] We employ the term *object mechanism* to designate mechanisms that depend on the dynamic type of a receiver, e.g., *virtual call* or *subtype test*.



Each box represents a unit of code, and arrows indicate compilation processes. In this example, a program written in two modules respectively in C# and Visual Basic is compiled using our compilation scheme. The third MSIL box represents a library which is referenced by the program.

Figure 1 – Compilation scheme

2.1 Compilation Scheme

Compilation schemes describe the transformation process from the source code to an executable. The scheme used in .NET and JAVA is particular in that it includes two steps. First, source-level languages like C# are compiled to a processor-agnostic intermediate language (which is called MSIL in the case of .NET). This intermediate language is then executed by the platform in either a compiled or interpreted manner. The first step involves compilers and linkers dedicated to specific languages to make them compatible with the platform, while the second enables the execution of the intermediate language for a specific architecture. We are concerned with the latter as it is the one that supports object-oriented features. In the case of .NET, this step is specified by the Common Language Infrastructure standard.

Well-known implementations such as CLR [MM00] and Mono [Mon] are based on a virtual machine architecture in order to support dynamic loading, and generally involve adaptive compilation [AFG+04]. We chose to disable dynamic loading, which allows us to implement global optimizations [Duc11b]. This choice may seem restrictive, however most firmwares do not require such a feature. Indeed, firmware updates consist in directly replacing the code in memory.

Our global compiler computes a conservative approximation of the behavior of the program using a type analysis algorithm. Its final implementation will support all .NET language features, except dynamic loading and introspection[2]. Indeed, it is not possible to detect the set of instantiated classes at compile-time when introspection is enabled. All of the other features are implemented at the moment, except genericity and exception handling. However, generics are supported in the type analysis so that we can analyze programs using them.

Compilation is performed ahead-of-time so that there is no negative impact on runtime. The compiler produces C code from the MSIL bytecode which is then compiled by the `gcc` port for Cortus APS3 (Figure 1). Compiling through C has been shown to be a viable strategy to implement JAVA on embedded systems [VB04] and allows us to focus on eliminating the overhead of object-oriented languages.

[2]In .NET, the `System.Reflection` namespace provides classes to introspect the program at runtime, and execute programs via predefined metaobjects.

2.2 Optimizations

Type analysis. Type analysis is a well-known optimization that works under the CWA, and Rapid Type Analysis (RTA) is the most commonly used algorithm [Bac97]. It constructs a call graph of the program, which allows us to identify the methods which are uncallable and remove it. It also approximates the dynamic types of the expressions in the program. Some virtual calls may be monomorphic (see Section 3.1), therefore they can be replaced with static calls. Moreover, some subtype tests may be solved at compile-time. The type analysis that we implemented is based on RTA and takes into account some .NET specific features such as array covariance and delegates (see Section 3).

Coloring. Coloring is an implementation technique for object mechanisms which extends the single subtyping implementation to multiple subtyping. This technique initially proposed by [DMSV89] has been proven to be efficient and well-suited in a global compilation framework [Duc11a]. We use this technique to compute our runtime structures. As well, we use the information generated by type analysis to remove some entries in these structures. Therefore method tables only contain polymorphic methods and type IDs for unsolved subtype test targets (see Section 4).

2.3 Garbage Collection and Libraries

Garbage collection is currently implemented using TinyGC [Tin], a mark-and-sweep garbage collector which provides a subset of the Boehm GC API [BW88]. This paper does not focuses on the efficiency of garbage collection. It will be considered in the future, as it has a significant impact on the runtime efficiency of the compiled program. Indeed, it would be preferable to use a precise GC rather than a conservative one, in order to reduce the memory overhead. Also, the use of a generational GC scheme has been shown to be well-suited for languages such as JAVA and C#, which create many short lived objects. An incremental GC scheme is also needed, as most embedded systems must satisfy real-time constraints.

For the low-level libraries, we implemented a significant subset of the .NET core class library and peripheral drivers for UART and GPIO. It has been possible to write this code entirely in C#. In the Cortus APS3 architecture, peripherals are mapped into memory, therefore it is possible to access them using the `unsafe` keyword, which allows C# programmers to use pointers. Similarly to C++, unsafe C# code may cause errors due to pointers such as memory leaks, therefore we only use it when necessary. As well, we implemented some interrupt handlers using static methods. At startup, trap vectors are initialized with the addresses of these methods.

3 Type Analysis

This section presents our type analysis which is based on Rapid Type Analysis [Bac97], a well-known algorithm, with optimizations which are specific to .NET. Various implementation issues are also discussed.

3.1 Rapid Type Analysis

Due to their dynamic nature, object mechanisms such as virtual call and subtype testing produce a substantial overhead at runtime. Given the information available

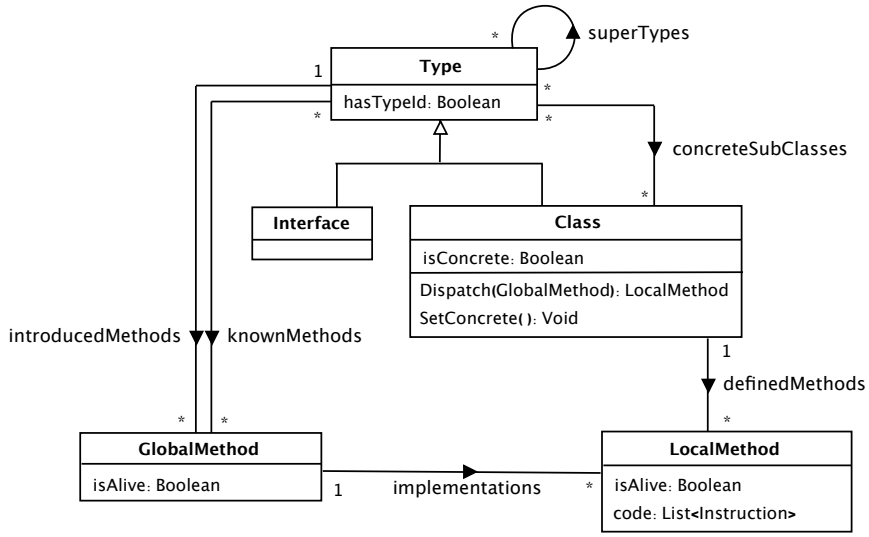


Figure 2 – Metamodel of types and methods

in a closed world, type analysis tries to approximate their behavior at compile-time. Typically, when a method is invoked, its implementation is determined by the dynamic type of the receiver. This type can be statically approximated with a precision dependent on the kind of analysis chosen. Similarly, some subtype tests can be resolved at compile-time as we can also approximate the actual type of the tested object. Our algorithm is based on RTA (Rapid Type Analysis) which computes two sets for instantiated classes and live methods [Bac97]. The approximation of each static type is the set of its instantiated subclasses. RTA mostly tries to resolve virtual calls, though it can be used to eliminate some subtype tests as well. The algorithm starts from the program entry point and constructs a call graph for the program. When a method call is reached, possible implementations for the method are deduced from the class hierarchy and the set of instantiated classes. They are thereby considered alive (i.e., callable) and their code is analyzed. Similarly, it is possible to know which fields are alive (i.e., accessed in the live code).

Metamodel. Unlike the original RTA, our representation of classes, fields and methods is based on a metamodel that removes multiple inheritance ambiguities which may occur with .NET interfaces [DP11]. It separates the identity of methods from their implementations. We use the terms global method to designate the former and local method for the latter (Figure 2). For instance, a method that is defined in a class and redefined in a subclass is represented by a single global method associated with two local methods. Each global method is introduced by a single class or interface[3].

Figure 2 shows a partial view of our metamodel. This partial view is restricted to types and methods, therefore it does not include fields. Both global and local methods have a flag which indicates whether they are alive. We can consider that live global methods may need an entry in the method tables, while live local methods

[3].NET and JAVA allow a method to be introduced by several interfaces which may be implemented by the same class. In this situation, we consider that they represent different global methods, therefore the method table of this class may have different entries filled by the same method address.

contain the live code. Fields also have a flag which indicates their liveness. Another flag shows if a given class is concrete, i.e., instantiated at least once in the live code. In the algorithm, this flag has to be set to `true` (all boolean flags are set to `false` by default in the metamodel) using the `SetConcrete` method, which propagates the instantiated class to the `concreteSubClasses`[4] of its supertypes. `Dispatch` returns the implementation for a given global method that a class knows. In addition, we add a flag so that we can recognize subtype test targets: since we use the Cohen technique extension to multiple subtyping [VHK97], this allows us to reduce the number of types which will need a unique identifier at runtime (see Section 4.1).

Algorithm 1: The base algorithm (RTA)

```
   Data: main, the entry point of the program
   Functions:
   StaticType(obj) returns the static type associated to the expression obj.
   GlobalMethod(meth, type) returns the global method associated to meth that type knows.
   LocalMethod(meth) returns the local method associated to meth.

   queue ← { LocalMethod(main) };
   while queue ≠ ∅ do
   |  current ← queue.Dequeue();
   |  foreach instr ∈ current.code do                          // parse current's code
   |  |  switch instr do
1  |  |  |  case new class                                          // instantiation
   |  |  |  |  if ¬ class.isConcrete then
   |  |  |  |  |  class.SetConcrete();                  // propagate to supertypes
   |  |  |  |  |  foreach globalMeth ∈ class.knownMethods
   |  |  |  |  |  when globalMeth.isAlive do
   |  |  |  |  |  |  class.Dispatch(globalMeth).SetAlive();
   |  |  |  |  |  end
   |  |  |  |  end
2  |  |  |  case obj isa type                                       // subtype test
   |  |  |  |  type.hasTypeId ← true;
   |  |  |  case obj.field                                          // field access
   |  |  |  |  field.isAlive ← true;
3  |  |  |  case meth()                                             // static call
   |  |  |  |  LocalMethod(meth).SetAlive();
4  |  |  |  case obj.meth()                                        // virtual call
   |  |  |  |  type ← StaticType(obj);
   |  |  |  |  globalMeth ← GlobalMethod(meth, type);
   |  |  |  |  globalMeth.isAlive ← true;
   |  |  |  |  foreach subClass ∈ type.concreteSubClasses do
   |  |  |  |  |  subClass.Dispatch(globalMeth).SetAlive();
   |  |  |  |  end
   |  |  |  endsw
   |  |  endsw
   |  end
   end
```

Base algorithm. RTA starts from the entry point. It analyzes the code of reachable methods and executes certain actions associated with specific expressions — namely *class instantiations*, *subtype tests*, *field accesses*, *static calls* and *virtual calls* (see Algorithm 1). `SetAlive` marks a local method as alive and enqueues it if it has not been previously analyzed.

[4]The property `concreteSubClasses` associates a given static type to the set of its potential dynamic types.

- Class instantiations cause the class to be marked as concrete and add the class to the `concreteSubClasses` of its supertypes (`SetConcrete`). If the class defines a local method which is associated with a live global method, it is marked as alive and enqueued (`SetAlive`).
- Subtype tests cause the target type to be marked as subtype test target (`hasTypeId`).
- Field accesses cause the field to be marked as alive (`isAlive`).
- Static calls cause the callee to be marked as alive and enqueue it (`SetAlive`).
- Virtual calls cause the set of dispatched local methods to be computed (`Dispatch`). Each method of this set is marked as alive and enqueued (`SetAlive`). If it is a virtual call, the associated global method is marked as alive too (`isAlive`).

Monomorphic call detection. An expression is said to be monomorphic when its value at runtime will always be of the same dynamic type. When the receiver is monomorphic, a virtual call always invoke the same method. Here, we use monomorphic in a wider meaning, that is when a virtual call always invoke the same method (even when the receiver is not monomorphic). The call graph resulting from RTA shows for each call site the set of local methods that can actually be invoked. When this set is reduced to a singleton, the call is monomorphic and can be implemented as a static call. Moreover, inlining can be performed when it does not increase code size, but we leave the decision to `gcc`.

Algorithm 2: Monomorphic call detection

```
Data: globalMeth the callee and type the receiver's static type
Result: localMeth the dispatched method, or null if call is not monomorphic
localMeth ← null;
foreach subClass ∈ type.concreteSubClasses do
    if localMeth = null then
        localMeth ← subClass.Dispatch(globalMeth);
    else if localMeth ≠ subClass.Dispatch(globalMeth) then
        return null;
    end
end
return localMeth;
```

Algorithm 2 describes a procedure which detects whether a given virtual call is monomorphic or not. Given the informations generated by Algorithm 1, if there is only one local method which can be dispatched, the call is considered monomorphic and can be implemented as a static call.

Dead code elimination. As a result of RTA, we know which methods are dead. Under the closed-world assumption, their code can be eliminated, which can be quite effective for programs that use large frameworks such as .Net or Java. Besides, reducing code size is a major concern of the embedded world. Global methods and fields that are dead do not need to be compiled. They can be eliminated from method tables and object layouts respectively. As well, live global methods that are never called in a polymorphic manner do not need an entry in the method tables, and abstract classes do not need such table at runtime.

Subtype test elimination. Subtype tests allow runtime checking of whether an object is typed by a specified type (called the *target type*) or not. This is typically needed for downcasts and for array assignments, which are unsafe in .NET and JAVA as covariant type overriding is allowed (see Section 3.2). We associate subtype check sites to the set of types that the tested object can take at execution. When all these types are subtypes of the target, the test is guaranteed to return true and can be removed. Conversely, when no subtype is contained within the set, we are guaranteed that the test returns false and we can replace it by the appropriate code (i.e., `throw` or `ldnull`). Moreover, the programmer can be warned at compile-time, which enhances software reliability.

Algorithm 3: Subtype test elimination

```
Data: obj the tested object and target the target type
Result: unsafe if test is unsafe, otherwise true or false
canBeTrue, canBeFalse ← false;
foreach subClass ∈ StaticType(obj).concreteSubClasses do
    if subClass ∈ target.concreteSubClasses then
        if canBeFalse then
            return unsafe;
        end
        canBeTrue ← true;
    else
        if canBeTrue then
            return unsafe;
        end
        canBeFalse ← true;
    end
end
return canBeTrue;
```

Algorithm 3 describes a procedure which detects whether a given subtype test can be resolved at compile-time or not. Given the informations generated by Algorithm 1, for each concrete type of the tested object, the test must always return the same result in order to be resolved at compile-time.

3.2 Array Covariance

Array covariance means that if class `B` is a subtype of `A`, it implies that `B[]` is a subtype of `A[]`. Assignments to elements of arrays are thus generally unsafe, as an array statically typed by `A[]` can actually be an instance of `B[]`, and a subtype test must be performed to ensure that the type of the assigned value is an instance of `B`.

Metamodel. Figure 3 shows how arrays are represented in our metamodel. When an array typed by `A[]` is instantiated, i.e., when the analysis reaches a `new A[]` expression, we invoke `SetElementType` on the instance of the metamodel which represents its element type, that is `A`. In a similar way to `SetConcrete`, this propagates `A` to the `elementSubTypes` of its supertypes. This allows the approximation of the dynamic type of an array given its static type.

Extended algorithm. Algorithm 4 adds a new action to perform in the base algorithm of RTA (see Algorithm 1). This action is associated with *array instantiations*, and marks the element type of the array (`SetElementType`).

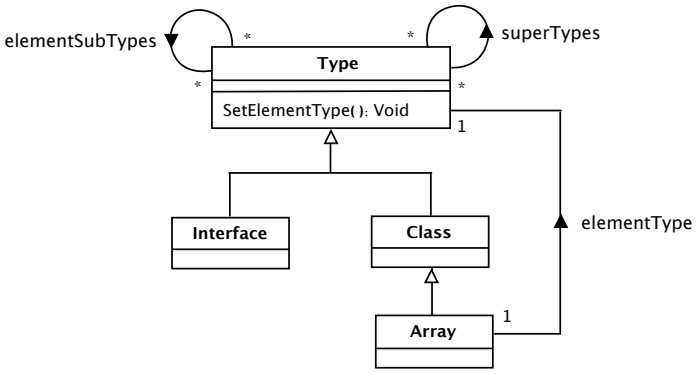


Figure 3 – Array support in metamodel

Algorithm 4: Extension of RTA for arrays

```
case new array                                          // array instantiation
  | array.elementType.SetElementType();                 // propagate to supertypes
```

Array store check elimination. Subtype test elimination does not apply directly here as we do not know the target type at compile-time (i.e., the element type of the array). We approximate it the following way: the element type of an array statically typed by `A[]` can be of any subtype `X` of `A` when `X[]` is marked as concrete. As both target and source have their types approximated, many combinations are possible, which reduces the chances of having all tests return the same result. However, if the array type is specific, this technique can eliminate many tests produced by array covariance. Algorithm 5 is similar to Algorithm 3. Given the informations generated by Algorithm 1 and 4, it detects whether the subtype test associated to an array access can be resolved at compile-time or not.

Algorithm 5: Array store check elimination

```
Data: obj the assigned object and array the static type of the assigned array
Result: unsafe if test is unsafe, otherwise true of false

canBeTrue, canBeFalse ← false;
foreach subClass ∈ StaticType(obj).concreteSubClasses do
  | foreach eltType ∈ array.elementType.elementSubTypes do
  |   | if subClass ∈ eltType.concreteSubClasses then
  |   |   | if canBeFalse then
  |   |   |   | return unsafe;
  |   |   | end
  |   |   | canBeTrue ← true;
  |   | else
  |   |   | if canBeTrue then
  |   |   |   | return unsafe;
  |   |   | end
  |   |   | canBeFalse ← true;
  |   | end
  | end
end
return canBeTrue;
```

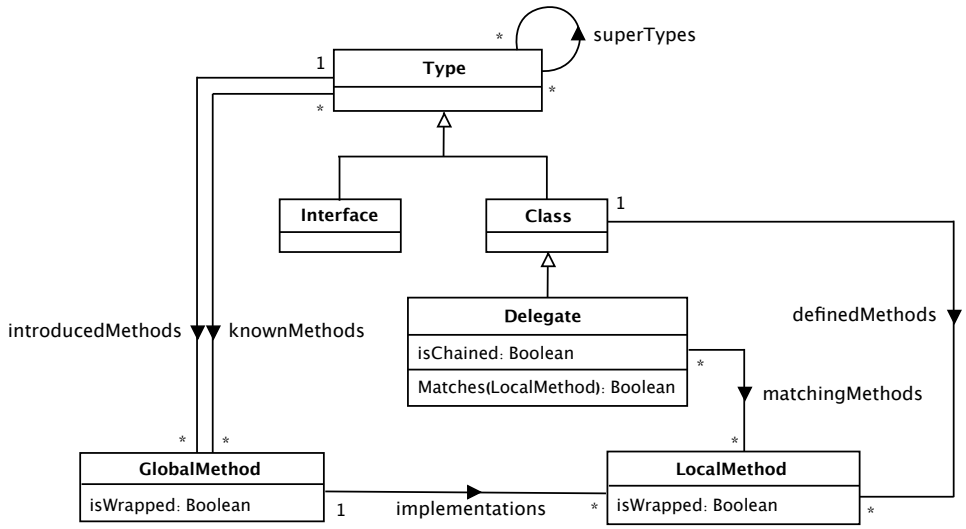


Figure 4 – Delegate support in metamodel

3.3 Generics

In .NET, generics are supported at the bytecode level, contrarily to JAVA which uses *type erasure*. Expressions involving run-time types such as `new T` are allowed, which must be considered when analyzing the bytecode. We compute the set of generic instances for each open generic type (a.k.a. formal type) in the program, as for arrays. When expressions such as `new T` are encountered, RTA executes the appropriate action for each generic instance of the open generic type. For `new T`, generic instances of `T` are marked as concrete (see Algorithm 1, lines 1 to 2) and their default constructor are marked alive (see Algorithm 1, lines 3 to 4). As a result, the call graph of the analyzed program is complete.

3.4 Delegates

Delegates allow programmers to encapsulate methods within objects, and they are typically used to define callback methods. A delegate is typed by a class which inherits `System.Delegate` and introduces a method named `Invoke`, which allows the wrapped method to be called with parameters of the right types. When a method is wrapped into a delegate, its signature must match the signature of `Invoke`[5]. Moreover, a delegate may be chained with other delegates of the same type, in which case calling `Invoke` implies several method calls with the same parameters (but different receivers).

Metamodel. Figure 4 shows how delegates are represented in our metamodel. Both global and local methods have a flag which indicates whether they are wrapped in delegates. For each delegate class, `Matches` checks if a given method matches the signature of the `Invoke` method.

[5]However, methods do not need to match the delegate's signature exactly. The return type may be covariant and parameter types may be contravariant.

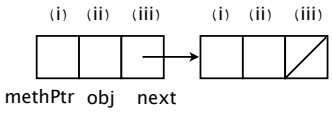


Figure 5 – Representation of two chained delegates

Extended algorithm. Algorithm 6 describes an extension of the base RTA (see Algorithm 1) which takes into account *delegates*. Methods which are wrapped into delegates are marked as alive (`SetAlive`) and wrapped (`isWrapped`). The action associated with *class instantiations* is extended. If the instantiated class introduces a local method which is associated with a wrapped global method, the local method is also marked as wrapped (`isWrapped`).

Algorithm 6: Extension of RTA for delegates

```
case new class                                                    // instantiation
    if ¬ class.isConcrete then
        ...  // same as the base algorithm
        foreach globalMeth ∈ class.knownMethods
        when globalMeth.isAlive do
            localMeth ← class.Dispatch(globalMeth);
            ...  // same as the base algorithm
            if globalMeth.isWrapped then
                localMeth.isWrapped ← true;
            end
        end
    end
case ldftn meth                                                   // ldftn
    localMeth ← LocalMethod(meth);
    localMeth.SetAlive();
    localMeth.isWrapped ← true;
case ldvirtftn obj.meth                                           // ldvirtftn
    type ← StaticType(obj);
    globalMeth ← GlobalMethod(meth, type);
    globalMeth.isAlive ← true;
    globalMeth.isWrapped ← true;
    foreach subClass ∈ type.concreteSubClasses do
        localMeth ← subClass.Dispatch(globalMeth);
        localMeth.SetAlive();
        localMeth.isWrapped ← true;
    end
```

Delegate representation. A delegate can be represented by a class which owns three fields (Figure 5): (i) a reference to the method which is wrapped into the delegate; (ii) a reference to the `this` argument to pass to the wrapped method (`null` if method is static); (iii) a reference to the next delegate in the chain to invoke (if any).

Delegate instantiation. A delegate is instantiated by passing (i) and (ii) to the constructor of its class. (ii) is computed using the bytecode instruction `ldftn` or `ldvirtftn`. Both load the method address, statically for the former. The latter is equivalent to a virtual call, except the method is not actually called. In other words, it is equivalent to `Dispatch` in the metamodel (Figure 2), and the method address is

accessed via a method table. It is possible to optimize these virtual dispatches in the same way we optimized virtual calls in the base algorithm (see Section 3.1). When the set of local methods that can actually be dispatched is reduced to a singleton, the address of the method can be directly accessed.

Chained delegates. It is possible to chain two delegates of the same type. This is done using the static method `Combine` of `System.Delegate`. Its returned value is typed by `System.Delegate`, therefore the compiler generates a cast to the appropriate delegate type. As a result, given a delegate class, we can determine at compile-time whether its instances may be chained (which allows the field `isChained` to be initialized in Figure 4). If not, we can remove the `next` pointer of its instances. Invoking a chained delegate causes to invoke each delegate in the chain. When the return type of this delegate is not `void`, it is the return value of the last delegate in the chain which is returned.

Delegate invocation. A delegate is invoked calling the `Invoke` method of its class. This method is annotated with the keyword `runtime`, which means it is implemented in the virtual machine. Therefore, we have to generate the appropriate code for this method. Given `RV` the return type of the delegate signature, the following code works for all classes of delegates:

```
RV Invoke(...)
{
    RV retval;

    // check if method is static
    if (obj == null)
        retval = (*methPtr)(...); // indirect call
    else
        retval = (*methPtr)(obj, ...); // indirect call

    // check if delegate is chained
    if (next != null)
        next.Invoke(...); // call next delegate in chain
    else
        return retval; // last delegate in chain
}
```

Delegate optimization. The above code is generated for the general case. However, given a specific delegate class, it is possible to generate more efficient code. For this purpose, we maintain a set of wrapped methods. This is done the following way: methods whose addresses are accessed using `ldftn` or `ldvirtftn` are considered wrapped. For each delegate class, wrapped methods that match the signature of `Invoke` may be called through this class of delegates. If all these methods are non-static, we can remove the first check. If they are all static, we can remove the `obj` field too. If there is only a single method that may be called, we can replace the indirect call by a direct call and remove the `methPtr` field. Moreover, the second check can be removed for classes of delegates which are never chained (see above).

Algorithm 7 describes a procedure which detects at compile-time whether a delegate invocation can be implemented as a direct call or not. Given the informations generated by Algorithm 1 and 6, if only one local method is both wrapped and matches the delegate's signature, the delegate call is monomorphic.

Algorithm 7: Check if delegate call can be direct

```
Data: delegate the class introducing Invoke, and wrappedMeths the set of wrapped methods
Result: localMeth the method to call directly, or null if the call is indirect
localMeth ← null;
foreach wrappedMeth ∈ wrappedMeths do
    if delegate.Matches(wrappedMeth) then
        if localMeth = null then
            localMeth ← wrappedMeth;
        else
            return null;
        end
    end
end
return localMeth;
```

Inlining. Under some circumstances, the code of the `Invoke` method may be inlined. We let `gcc` choose whether to inline or not. For classes of delegates where `Invoke` consists in a single direct call, it is possible to replace calls to `Invoke` with a direct call to the wrapped method. In this case, the delegate class and its instances can be removed from the program (i.e., code, instantiations and invocations) if the delegate fields are never accessed in the live code. If the wrapped method is non-static, the delegate instance can be replaced in the stack with the `obj` field.

3.5 Analysis Details

Static types and evaluation stack. In the C# code, static types are annotations of names. As the .NET virtual machine is stack-based, when we analyze the bytecode, we have to track the types on the stack in order to retrieve the static types of all operands. This is done by implementing the verification algorithm described in the Common Language Infrastructure standard [MR04]. However, we are using it for type propagation purposes, not bytecode verification, since we make the assumption that the high-level compiler generates correct bytecode.

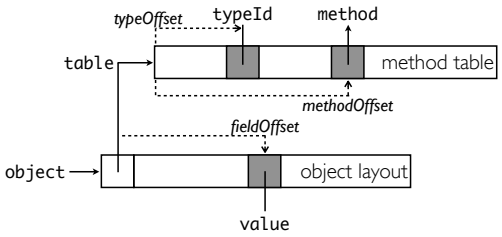
Intraprocedural control flow analysis. In addition to RTA, we use an intraprocedural control flow analysis. It propagates the concrete types induced by literals and instantiations (`new`) in the context of a single method. We use this information when we construct the call graph of the program, which reduces its size. As a result, our type analysis is more precise than RTA only.

Black boxes. Some keywords (e.g., `internalcall`, `runtime` and `native`) specify that a method is directly implemented in the virtual machine. This is typically used by low-level methods in a system library, which are not portable. As a result, some code is located in the virtual machine itself. There is an issue in the case where we do not have access to this code. For example, performing type analysis on programs that target a proprietary implementation such as CLR does not permit us to have complete knowledge of the code, which is a requirement for the optimizations we implement. This is however a problem in our preliminary study, not the final implementation, as we will have our own libraries and therefore full access to the code when the compiler will be operational. In the preliminary study, internal methods can be considered as *black boxes* as we do not have access to their implementation and we are only

```
// field access
load [object + #fieldOffset], value

// virtual call
load [object + #table], table
load [table + #methodOffset], method
call method

// subtype test
load [object + #table], table
load [table + #typeOffset], typeId
comp typeId, #targetId
bne #fail
// succeed
```



The code sequences are expressed in the intuitive pseudo-code proposed by [Dri01]. The diagram depicts the corresponding object representation. Pointers and pointed values are in roman type with solid lines, and offsets are italicized with dotted lines. Each mechanism relies on a single invariant offset.

Figure 6 – Code sequences and object representation

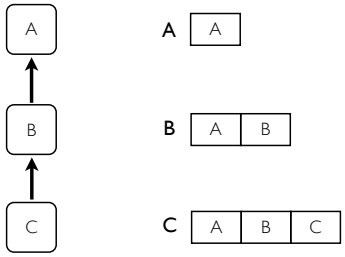
concerned about their output, that is their return type. Typically, if a subclass `B` of `A` is instantiated in a black box and returned under the static type `A`, `B` must be considered alive, which is not assumed by our type analysis in the case where `B` is never instantiated in the live code. A solution is to consider that all non-abstract subclasses of the black box return type are instantiated. As a result, type analysis loses in precision but remains sound. There is however a huge loss of precision when the return type is a universal supertype such as `Object` or `Array`. In this case, we detect the casts applied to the returned object and we consider that all non-abstract subclasses of the cast targets are instantiated.

4 Implementation Technique

Implementation techniques determine the representation of objects, that is the data structures that support object-oriented mechanisms — namely virtual call, subtype test and field access. This section presents the technique that we chose, which is known as coloring, and extends the single subtyping implementation to multiple subtyping [Duc11a].

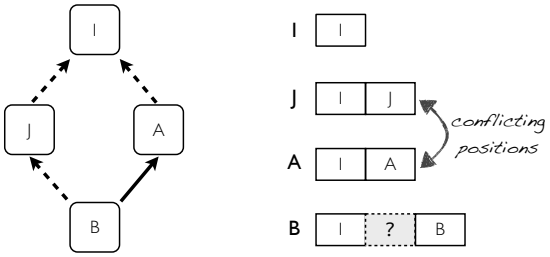
4.1 Single Subtyping Implementation

In statically typed languages, an object is typically implemented as a table of its fields with a pointer to the method table which is associated with its class. The layout of these tables depends on the implementation technique chosen. In single subtyping, the implementation relies on *reference invariance* and *position invariance*, which means that the code sequences generated to support object mechanisms does not depend on the *static type* nor the *dynamic type* of the receiver (Figure 6). Object layouts and method tables are straightforward extensions of those of its direct superclass (Figure 7). As a result, this technique is compatible with dynamic loading.



The figure at the left shows a hierarchy of classes, which are represented by boxes. Solid arrows represent class inheritance. The figure at the right shows the associated method tables, where boxes represent the group of methods introduced by each type.

Figure 7 – Single subtyping implementation

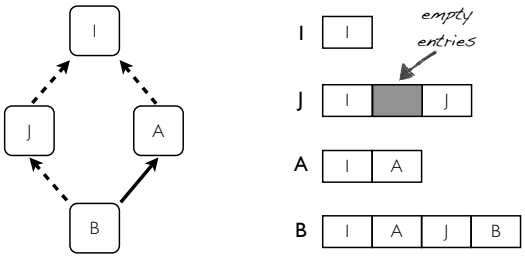


The figure at the left shows a hierarchy of classes and interfaces. Solid arrows represent class inheritance, and dashed arrows represent interface subtyping. The figure at the right shows the associated method tables, where boxes represent the groups of methods introduced by each type.

Figure 8 – Conflicting positions

Subtype tests. Subtype testing can be implemented using the technique proposed by [Coh91], which consists assigning both a unique ID and an invariant position in the method table to each class. It is guaranteed that an object `x` is an instance of the class `A` if and only if the method table of `x` contains the type ID of `A` at the position associated with `A`. In our implementation, we use method tables to hold both methods and type IDs (Figure 6). There is an ambiguity in the case where a method is located at an address `n` which is also a type ID, as some subtype tests may return true erroneously. A solution is to choose non-even values for type IDs, as addresses are aligned. Some implementations of the Cohen technique need to check the bounds of the table, as the position associated with a type ID may be greater than the upper bound of the actual table that the tested object points to. This check is eliminated in our case, as method tables are consecutive in memory and the largest one is placed in the final position.

Array store checks. Some additional data at runtime is needed to perform array store checks (see Section 3.2). The code sequence for this mechanism is similar to a subtype test (Figure 6) except that `#typeOffset` and `#targetId` must be computed at runtime. We chose to put this information directly in the array instance rather than in its method table. Besides, when array store checks are eliminated in the type analysis, this information can be removed.



The figure at the right shows the associated method tables, where boxes represent the groups of methods introduced by each type. The hole induced by the conflict between J and A is placed in the method table of J, which is inexistent at runtime as interfaces cannot be instantiated.

Figure 9 – Coloring implementation

4.2 Coloring Implementation

Though multiple inheritance is not fully supported in .NET and JAVA, multiple subtyping is possible through special types called *interfaces* which can only introduce *abstract methods*. In this context, fields are only introduced by classes. As a result, the single subtyping implementation works for object layouts. In contrast, methods can be introduced by unrelated classes or interfaces in such a way that they would have different positions in the single subtyping implementation (Figure 8). In other words, the position invariant cannot be ensured for methods and interface IDs. The general idea of coloring consists in inserting empty entries in order to maintain position invariance (Figure 9). This technique works under the CWA and has the same efficiency as single subtyping, except space cost. In this respect, the number of holes has to be minimized. Coloring can be described as the technique that keeps single subtyping invariants at minimal space cost. It has been separately proposed for the three basic mechanisms (i.e., virtual call, field access and subtype test) [DMSV89, PW90, VHK97]. Like minimal graph coloring, the coloring problem considered here has been proven to be NP-hard in the general case, but efficient heuristics have been proposed.

Interfaces. To some extent, it should be possible to take advantage of the abstract nature of interfaces in order to reduce the number of holes. Like abstract classes, interfaces are never instantiated, which means their method tables do not exist at runtime. They are only computed so that they can be extended in the method tables of the instantiated subclasses, which are concrete. Accordingly, holes introduced in the abstract tables can be filled in the concrete tables. It should be advantageous to consider this when performing coloring, as space cost may be reduced (Figure 9). Moreover, we can consider classes that are never instantiated in the live code as abstract too.

Unidirectional coloring. The number of holes induced by coloring can be reduced using positive and negative offsets on the method tables. This technique is known as bidirectional coloring [PW90]. However, the Cortus APS3 does not support negative offsets in load instructions, therefore an additional subtraction would be needed for some accesses in the method tables. We chose unidirectional coloring, which only uses positive offsets.

4.3 Optimizations

Type analysis allowed us to remove some entries from the method tables. As explained in Section 3, methods that are never called polymorphically do not need an entry in the method tables. Similarly, types that are never used as subtype test targets in the live code do not need a type ID. Therefore method tables only contain necessary data plus some holes, which would have not been possible under the OWA. The following optimizations concern what remains in these structures.

Filling empty entries. As the drawback of coloring is the insertion of holes in the method tables, we propose using these empty entries to store static data. The only constraint is to maintain subtype test integrity, which means that a hole cannot be filled with a value that is not even. For example, static fields can be used to fill empty entries in the method tables. We chose to fill holes with objects as they are pointers and therefore even values. This optimization offers two advantages: (i) reduction of the number of holes; (ii) locality in the case where a static field `f` can fill a hole in the method table of a class that knows `f`.

Merging colors. A class named `A` that implements an interface `I` must provide a definition for the methods that `I` knows. A priori, a method `m` introduced in `I` designates the same method that the method `m` defined in `A`, therefore they share the same entry in the method tables. However, it is possible to provide separate definitions for these methods in the class definition of `A`. This feature is known as explicit overriding in .NET [MR04] and explicit interface implementation in C# [JPS07]. As a result, `A` must provide a definition for $\texttt{m}_\texttt{I}$, but introduces its own method $\texttt{m}_\texttt{A}$ as side effect. Both methods share the same implementation, except when there is an explicit implementation of $\texttt{m}_\texttt{I}$ in `A`. Under the CWA, it is possible to know whether $\texttt{m}_\texttt{I}$ and $\texttt{m}_\texttt{A}$ share the same implementation in all classes or not. If it is the case, both entries in the method tables contain a pointer to the same code (see Section 3.1), and it is possible to unify them. Accordingly, $\texttt{m}_\texttt{I}$ and $\texttt{m}_\texttt{A}$ share the same entry (i.e., color) in the method tables when possible.

5 Tests and Results

In this section, we evaluate the global techniques presented in this paper on programs written in C#.

5.1 Target Programs

We implemented the compiler described in Section 2, including our extensions for RTA (see Section 3) and the implementation technique described in Section 4. However, having a functional compiler is not sufficient to execute real programs on our specific architecture, that is Cortus APS3. Indeed, the .NET framework provided by Microsoft is not portable as it includes some internal calls to the virtual machine and operating system (see Section 3.5). As a result, we need to provide our own framework and re-implement the libraries referenced by the target programs. This can be quite tedious, so we chose a different approach. Instead of measuring execution time and memory consumption at runtime, our compiler counts the number of optimized entities (e.g., virtual calls, subtype tests) for a given program and shows the results with and without optimizations.

number of	classes			interfaces		
	program	loaded	*red.*	program	loaded	*red.*
Clock	408	201	*51%*	16	11	*31%*
ServiceClient	493	263	*47%*	24	14	*42%*
Compiler	5361	1657	*69%*	451	204	*55%*
Gateway	13653	2797	*79%*	1019	276	*72%*

(a) Class and interface loading

Each subtable corresponds to a specific kind of type (class or interface). For each subtable, the first column shows the number of types in the whole hierarchy (program and libraries), whereas the second column shows the number of types loaded in our compiler according to RTA.

number of	instantiated		supertypes	
	classes	arrays	classes	interfaces
Clock	93	21	2.3	0.2
ServiceClient	145	20	1.9	0.2
Compiler	841	164	3.7	1.6
Gateway	1369	305	3.4	1.4

(b) Instantiated classes and average number of supertypes

The first subtable shows the number of instantiated types. The second subtable shows the average number of supertypes per type (direct and indirect).

Figure 10 – Class hierarchy

Our target programs can be split in two categories:

- programs based on the .NET Micro Framework, a smaller version of the .NET Framework dedicated to embedded systems,
- programs based on the standard .NET Framework — these programs are bigger and use consequent libraries.

Experiments showed that results are homogeneous for a given framework, therefore we only show them for two programs per framework. In the first category, Clock and ServiceClient are samples of the .NET Micro Framework SDK. In the second category, Compiler is our implementation of the Common Language Infrastructure and Gateway is a wireless access point based on the standard .NET Framework.

5.2 Results

As explained in Section 3.5, our current solution for the black box issue reduces the efficiency of type analysis, which has to be considered before discussing measurements. Also, the way of writing programs (use of polymorphism for example) and the size of the framework classes may have a significant impact on results (see Section 5.3).

Class hierarchy. Our compiler loads the model of classes which are instantiated in the live code and the model of their supertypes (direct or indirect). Figure 10(a) compares the entire program with what is actually loaded. The difference results from the use of libraries in .NET, which contain more classes than necessary for one single program. The small numbers of loaded interfaces and super-interfaces per type (Figure 10(b)) means that most classes do not implement interfaces.

number of	instance fields			static fields		
	loaded	live	*red.*	loaded	live	*red.*
Clock	434	316	*27%*	630	83	*87%*
ServiceClient	596	488	*18%*	649	80	*88%*
Compiler	4936	3066	*38%*	7489	918	*88%*
Gateway	11670	7600	*35%*	15271	1878	*88%*

(a) Dead field elimination

Each subtable corresponds to a specific kind of field (instance or static). For each subtable, the first column shows the number of fields in the loaded types whereas the second column shows the number of live fields.

number of	non-virtual methods			virtual methods					
				global			local		
	loaded	live	*red.*	loaded	live	*red.*	loaded	live	*red.*
Clock	1105	513	*54%*	272	79	*71%*	353	114	*68%*
ServiceClient	1310	737	*44%*	358	121	*66%*	542	173	*68%*
Compiler	14546	6869	*53%*	4032	1047	*74%*	7640	2331	*69%*
Gateway	28815	15001	*48%*	7200	1974	*73%*	14070	4772	*61%*

(b) Dead method elimination

Each subtable corresponds to a specific kind of method (non-virtual or virtual). Non-virtual methods include both instance and static methods. For virtual methods, we distinct global and local methods according to the definition given in Section 3.1. For each subtable, the first column shows the number of methods in the loaded types, whereas the second column shows the number of live methods.

bytecode	loaded	live	*red.*
Clock	41KB	25KB	*39%*
ServiceClient	78KB	85KB	*26%*
Compiler	1142KB	629KB	*45%*
Gateway	2519KB	1481KB	*41%*

(c) Bytecode size reduction

For each program, the first column shows the size of bytecode in the loaded types whereas the second column shows the size of live bytecode. Sizes are given in kilobytes.

Figure 11 – Dead code elimination

Dead code elimination. Figures 11(a) and 11(b) show that only a few fields and methods in the loaded types are alive. As explained in Section 3, dead members can be eliminated as we do not need to compile them. Also, code size reduces as the number of methods decreases (Figure 11(c)).

Delegates. Figure 12 shows that only a few delegate classes are actually both instantiated and invoked. Delegates that are instantiated but not invoked can be removed from the program. Conversely, when a delegate which is not instantiated is invoked, its value is `null`. In this respect, we do not need to compile the call site. Moreover, we can warn the programmer that a null check should be inserted prior to the call site. Figure 12 also shows that we can remove the first check and the chain check in most `Invoke` methods (see Section 3.4). In some cases, we can replace the indirect call by a direct one. When all three optimizations are possible, we can remove the delegate class from the program and replace the call to `Invoke` with a direct call to the wrapped method.

number of	delegate classes			Invoke methods			
	loaded	live	red.	opt1	opt2	opt3	all opts
Clock	21	14	*33%*	10	9	2	1
ServiceClient	11	7	*36%*	6	4	1	0
Compiler	60	31	*48%*	24	29	10	10
Gateway	170	88	*48%*	37	52	15	9

The first subtable shows the number of delegate classes for a given program. The first column shows those which are loaded in our compiler whereas the second column shows those which are alive, i.e. instantiated and invoked. The second subtable considers the optimizations that can be applied to the `Invoke` methods of these live delegate classes. `opt1` stands for eliminating the first check in `Invoke`. `opt2` stands for eliminating the chain check, whereas `opt3` stands for replacing the indirect call by a direct call. The last column shows the number of `Invoke` methods for which we can apply all three optimizations.

Figure 12 – Delegates

Invocation sites. When object mechanisms are resolved at compile-time, they can be more efficiently implemented, as indirections in the method tables can be avoided. Figure 13(a) shows that most method calls are static. This is due to the fact that C# considers methods as non-virtual by default, unless the `virtual` keyword is explicitly used. Therefore, polymorphism is less used than in languages like Java. Concerning virtual calls, most of them are actually monomorphic and can be implemented as static calls. Figure 13(b) shows that most subtype checks due to array covariance can be eliminated, in contrast to those due to downcasts. This can be explained by the fact that only a few arrays are instantiated in the live code (Figure 10(b)).

Runtime structures. Figures 14(a) and 14(b) show that runtime structures are accordingly small, as object layouts and method tables respectively contain live fields and live polymorphic methods plus a few type IDs.

Coloring. Figure 15(a) shows which entities in the program need to be colored. It shows that method tables only contain a few type IDs (those for types which are targets of one or many unsafe subtype tests) and a few methods (those which are called in a polymorphic manner). Figure 15(b) shows that there is a small proportion of holes in the method tables. Moreover, a significant part of them can be filled using static references.

5.3 Discussion

The results show a noticeable reduction of the programs. Half of the types can be ignored at compile-time and about half of the methods do not need to be compiled, which reduces the code size.

Frameworks. The frameworks used by the target programs have a significant influence on the results. In fact, these frameworks are generally bigger than the programs themselves. To some extent, running RTA on two programs which use the same framework can be viewed as analyzing the same code but starting from two different entry points. Monomorphic call detection is less efficient for big programs which target the standard .Net framework. Using an interprocedural control flow analysis might reduce the gap resulting from the program size.

number of	static calls	virtual calls		
		all	poly.	*red.*
Clock	1317	187	36	*81%*
ServiceClient	3014	602	78	*87%*
Compiler	33940	8812	3445	*61%*
Gateway	83592	18037	8269	*54%*

(a) Method calls

Each subtable describes a specific kind of method call (static or virtual). For virtual calls, the first column shows the number of virtual calls in the live bytecode whereas the second shows the number of polymorphic calls. Monomorphic calls can be implemented as static calls.

number of	explicit subtype tests			array store checks		
	all	unsafe	*red.*	all	unsafe	*red.*
Clock	81	80	*1%*	29	0	*100%*
ServiceClient	84	82	*2%*	199	0	*100%*
Compiler	2294	2200	*4%*	2161	79	*96%*
Gateway	5254	5009	*5%*	4176	100	*98%*

(b) Subtype tests

Each subtable describes a specific kind of subtype test (explicit or due to array covariance). For each subtable, the first column shows the total number of subtype tests whereas the second column shows the number of unsafe tests. Safe tests can be removed from the program.

Figure 13 – Invocation sites

number of entries	total			average	
	loaded	needed	*red.*	loaded	needed
Clock	785	455	*42%*	3.8	4.3
ServiceClient	745	579	*22%*	3.1	3.8
Compiler	9278	6019	*35%*	5.2	6.3
Gateway	18665	10751	*42%*	5.4	6.3

(a) Object layouts

This table describes the number of entries in the object layouts (one layout per non-abstract class). For each subtable, the first column considers the loaded program, whereas the second column shows the number of entries which are actually needed, i.e., live fields of concrete classes.

number of entries	total			average	
	loaded	needed	*red.*	loaded	needed
Clock	1969	403	*80%*	9.6	4.3
ServiceClient	2283	584	*74%*	9.4	4.0
Compiler	39310	12346	*67%*	22.4	13.7
Gateway	74915	26428	*65%*	21.8	19.3

(b) Method tables

This table describes the number of entries in the method tables. For each subtable, the first column considers the loaded program, whereas the second column shows the number of entries which are actually needed, i.e. live polymorphic methods and unsafe subtype test targets.

Figure 14 – Runtime structures

number of	methods			types		
	loaded	colored	*red.*	loaded	colored	*red.*
Clock	272	14	*95%*	223	37	*84%*
ServiceClient	358	13	*96%*	297	42	*86%*
Compiler	4032	320	*92%*	2025	516	*75%*
Gateway	7200	631	*91%*	3378	1016	*70%*

(a) Colored entities

Each subtable respectively describes the number of global methods and types which need an entry in the method tables. They respectively correspond to methods called from polymorphic call sites, and unsafe subtype test targets.

number of	methods	type IDs	holes	static refs
Clock	363	38	2	35
ServiceClient	543	39	2	41
Compiler	8585	2761	1000	492
Gateway	17213	5045	4170	1186

(b) Entries in method tables

For each program, the first column shows the number of method entries in the method tables. The second column shows the number of type IDs, and the third column shows the number of holes induced by coloring. The last column shows the number of static fields which are typed by a reference type, and can be used to fill holes in the method tables.

Figure 15 – Coloring

Polymorphism. Using C#, only a few method calls are virtual. Of these virtual calls, most are considered monomorphic by RTA, which means they can be implemented as static calls. This brings us back to the efficiency of procedural languages like C. We can say that the language resulting from our compilation process is less object-oriented than the high-level language, which is C# in the case of this experiment. Most method calls are static, and most array accesses do not need a subtype test due to covariance. However, most explicit subtype tests remain.

Coloring. We find only a few holes in the method tables, which can be explained by the fact that most classes do not implement interfaces and that the average size of the method tables is relatively small. In the worst case, we can fill a significant subset of these holes with static fields.

6 Related Work

This section presents related works. The first section presents some techniques which could be considered for implementation, whereas the second section shows an overview of existing implementations for Java and .Net that target embedded systems.

6.1 Related Techniques

Type analyses. Many type analysis algorithms exist, with their accuracy which depends on their cost. Control Flow Analysis (CFA) is an interprocedural analysis which gives better results than RTA, but its cost is considerably higher [Shi91]. However, scalable implementations of CFA have been proposed [Pro02]. As well, some intermediate solutions between RTA and CFA have been explored, such as eXtended

Type Analysis (XTA) [TP00]. Whereas RTA maintains a single set of concrete types for the whole program, XTA maintains a separate set for each method and field.

Implementation techniques. Some implementations are based on binary tree dispatch (BTD) rather than method tables [ZCC97]. The efficiency of BTD relies on conditional branch prediction. The PRM compiler has been used as an experimental platform to test various implementations for object-oriented languages [DMP09]. BTD and coloring were jointly tested, and their combination were shown to be the best choice under the closed-world assumption for processors equipped with branch prediction. However, branch prediction is not implemented in low-power processors such as the Cortus APS3 because it increases power consumption. Moreover, BTD tends to increase code size, which is inappropriate for small embedded systems. Therefore we do not use this technique. However, we implement monomorphic calls with static calls, which corresponds to a BTD of depth 0. Accordingly, our implementation can be described as MC-BTD_0-G in [DMP09].

Adaptive compilation. Popular virtual machines such as Sun's HotSpot and Microsoft's CLR use adaptive compilation in order to be efficient in the context of dynamic loading [AFG+04]. The techniques they use are not applicable in our context of static compilation. However, some techniques that we use could be tested in an adaptive compilation framework. Indeed, [IKY+00] proposes to use global optimizations under the OWA and to use some backup code in the case where initial hypotheses become invalidated at runtime. For instance, some virtual calls can be devirtualized until the assumption that they are monomorphic becomes invalidated. When it happens, the compiler performs *code patching* by substituting the optimized code by the backup code.

6.2 Embedded Runtime Systems

There is a large number of runtime systems that target embedded systems, especially for JAVA.

Interpreters. Sun's KVM [KVM00] and Microsoft's TinyCLR are based on an interpreter — TinyCLR is the execution engine behind the .NET Micro Framework [TM07]. Sun's JavaCard targets low-end embedded systems such as smart cards and converts the JAVA bytecode to a more compact format prior to execution [BBE+99]. It does not supports dynamic loading.

Dynamic compilers. The KVM has been replaced by Sun's CLDC HotSpot, which compiles *hot spot* methods at runtime [CLD05]. There is little information on its implementation. Armed E-Bunny [DMT05] and Dalvik VM [CB10] — the execution engine for Google's Android platform — also use a mix between interpretation and adaptive compilation. Microsoft's Compact Framework targets high-end embedded systems, and is based on the CLR which only uses dynamic compilation [WSW+02]. As the generated code is placed in RAM, a technique called *code pitching* consists of removing a subset of this code when there is a lack of memory.

Ahead-of-time compilers. Sun's CLDC HotSpot and Microsoft's CLR can also support ahead-of-time compilation. However, the optimizations proposed in this paper are not directly applicable in their case, as they must support dynamic loading. JamaicaVM uses ahead-of-time compilation, except for dynamically loaded code

which is interpreted [Sie02]. Similarly, Excelsior JET uses ahead-of-time compilation and dynamic compilation for the code which is dynamically loaded [MLG+02]. Excelsior JET also proposes a global optimization mode where RTA and dead code elimination are used.

Similar implementations. Type analysis and coloring are also implemented in FLEX [AR03], which includes various optimizations for Java programs, and FijiVM, an ahead-of-time compiler for Java which generates C code and uses 0-CFA rather than RTA [PZV09]. However, coloring is only used for interface methods. Our work is different in that language-specific features such as array covariance are considered in the type analysis (see Section 3.2). Moreover, we implement all virtual methods in the same table using coloring, and use static references to fill holes.

Other implementations. Proprietary implementations such as Aonix's Perc Pico [Aon] and I2ST's MicroJVM [Mic] claim to execute Java with the same efficiency as optimized C code. It would be interesting to have more details on these implementations, as there is little information available.

7 Conclusion and Future Work

This paper describes the global techniques that we use in our Msil-to-C static compiler, which targets low-end embedded systems. The type analysis that we use is an extension of RTA [Bac97] which takes into account *array covariance*, *generics* and *delegates*. It is used jointly with an intraprocedural control flow analysis. Runtime structures are computed using the coloring technique [Duc11a]. Our coloring implementation takes advantage of the information generated by RTA. Indeed, runtime structures contain only necessary data such as polymorphic methods and type IDs for subtype test targets.

Our experimental results show a noticeable reduction of the programs under the CWA, therefore it confirms that global compilation is an appropriate strategy for implementing object-oriented languages on embedded systems. A significant amount of the code size can be removed, and most virtual calls can be replaced with static calls. Moreover, array covariance has almost no overhead and delegate calls can be as efficient as static calls in some circumstances. Array store check elimination should be considered for Java programs as well.

The coloring implementation technique offers the efficiency of the single subtyping implementation. The drawback of this technique being the insertion of empty entries in the method tables, it can be attenuated by replacing these entries with some static fields when possible. Since interfaces cannot introduce fields, there are no empty entries in the object layouts. Therefore coloring is an appropriate implementation technique for .Net and Java as multiple subtyping concerns only interfaces.

Future Work

Further work will implement generics by taking advantage of the CWA. There are two common implementation schemes for generics; one consists of specializing the code among the instantiations of a generic class (heterogeneous, like in C++) whereas the other is based on *type erasure* and prevents performing operations that depend on the runtime type (homogeneous, as in Java) [OW97]. In the CLR, generics are implemented using a mixed scheme which tries to combine the benefits of both implementations [KS01]. So far, our extended RTA takes generics into account in order to make the call graph complete. It remains to exploit the results of the type analysis in order to determine which generic instances must be compiled, and according to which implementation scheme.

We will also optimize null checks, range checks, and implement exception handling. In order to add multithread support, we may use FreeRTOS [Fre] as our runtime does not include any operating system yet. We will also investigate an efficient garbage collection scheme that satisfies real-time constraints.

We chose RTA as a type analysis algorithm because it gives very good results at low cost. An interprocedural *control flow analysis* like CFA might however give better results, especially for delegates, and will be considered for implementation [Shi91].

References

[AFG+04] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proc. of the IEEE, 93(2), 2005. Special issue on program generation, optimization, and adaptation*, 2004. doi:10.1109/JPROC.2004.840305.

[Aon] http://www.atego.com/products/aonix-perc-pico/.

[AR03] C. S. Ananian and M. C. Rinard. Data size optimizations for Java programs. In *LCTES '03*, pages 59–68. ACM, 2003. doi:10.1145/780732.780741.

[Bac97] D. F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, December 1997.

[BBE+99] M. Baentsch, P. Buhler, T. Eirich, F. Höring, and M. Oestreicher. JavaCard - from hype to reality. *IEEE Concurrency*, pages 36–43, October 1999. doi:10.1109/4434.806977.

[BW88] H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, pages 807–820, September 1988. doi:10.1002/spe.4380180902.

[CB10] B. Cheng and B. Buzbee. A JIT compiler for Android's Dalvik VM. In *Google I/O '10*. Google, 2010.

[CLD05] The CLDC HotSpot implementation virtual machine. White paper. Sun Microsystems, Inc. 2005.

[Coh91] N. H. Cohen. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, pages 626–629, 1991. doi:10.1145/115372.115297.

[DMP09] R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *OOPSLA '09*, pages 41–60. ACM, 2009. doi:10.1145/1640089.1640093.

[DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA '89*, pages 211–214. ACM, 1989. doi:10.1145/74878.74900.

[DMT05] M. Debbabi, A. Mourad, and N. Tawbi. Armed E-Bunny: a selective dynamic compiler for embedded java virtual machine targeting ARM processors. In *SAC '05*, pages 874–878. ACM, 2005. doi:10.1145/1066677.1066876.

[DP11] R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. *Sci. Comput. Program.*, pages 555–586, July 2011. doi:10.1016/j.scico.2010.10.006.

[Dri01] K. Driesen. *Efficient Polymorphic Calls*. Kluwer Academic Publishers, 2001.

[Duc11a] R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. *Softw., Pract. Exper.*, pages 627–659, May 2011. doi:10.1002/spe.1022.

[Duc11b] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comput. Surv.*, pages 18:1–18:48, April 2011. doi:10.1145/1922649.1922655.

[Fre] http://www.freertos.org/.

[IKY+00] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *OOPSLA '00*, pages 294–310. ACM, 2000. doi:10.1145/354222.353191.

[JPS07] J. Jagger, N. Perry, and P. Sestoft. *C# Annotated Standard*. Morgan Kaufmann, 2007.

[KM10] D. Kerr-Munslow. Advantages and pitfalls of moving from an 8 bit system to 32 bit architectures. In *ERTS^2 '10*, 2010.

[KS01] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI '01*, pages 1–12. ACM, 2001. doi:10.1145/378795.378797.

[KVM00] J2ME building blocks for mobile device. White paper. Sun Microsystems, Inc. 2000.

[Mic] http://www.is2t.com/.

[MLG+02] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin. Overview of Excelsior JET, a high performance alternative to Java virtual machines. In *WOSP '02*. ACM, 2002. doi:10.1145/584369.584387.

[MM00] E. Meijer and J. Miller. Technical overview of the Common Language Runtime. Technical report, Microsoft Research, 2000.

[Mon] http://www.mono-project.com/Mono:Runtime.

[MR04] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure annotated standard*. Addison-Wesley, 2004.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *POPL '97*, pages 146–159. ACM, 1997. doi:10.1145/263699.263715.

[Pro02] C. W. Probst. *A Demand Driven Solver for Constraint-Based Control Flow Analysis*. PhD thesis, Universität des Saarlandes, October 2002.

[PW90] W. Pugh and G. E. Weddell. Two-directional record layout for multiple inheritance. In *PLDI '90*, pages 85–91. ACM, 1990. doi:10.1145/93542.93556.

[PZV09] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *JTRES '09*, pages 110–119. ACM, 2009. doi:10.1145/1620405.1620421.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-Order Languages -or- Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[Sie02] F. Siebert. Bringing the full power of Java technology to embedded realtime applications. In *MSy '02*, 2002.

[Tin] http://tinygc.sourceforge.net/.

[TM07] D. Thompson and C. Miller. Introducing the .NET Micro Framework. Technical report, Microsoft Corporation, 2007.

[TP00] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00*, pages 281–293. ACM, 2000. doi:10.1145/354222.353190.

[VB04] A. Varma and S. S. Bhattacharyya. Java-through-C compilation: An enabling technology for Java in embedded systems. In *DATE '04*. IEEE Computer Society, 2004.

[VHK97] J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. In *OOPSLA '97*, pages 142–157, 1997. doi:10.1145/263700.263730.

[WSW+02] A. Wigley, M. Sutton, S. Wheelwright, R. Burbidge, and R. Mcloud. *Microsoft .NET Compact Framework: Core Reference*. Microsoft Press, 2002.

[ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *OOPSLA '97*, pages 125–141. ACM, 1997. doi:10.1145/263698.263728.

About the authors

Olivier Sallenave is Research and Development Engineer at Cortus, a company which designs and licenses ultra low-power 32-bit processor cores. He is currently completing his PhD thesis at the University of Montpellier. This thesis focuses on the efficient implementation of object-oriented languages for low-end embedded systems. Contact him at olivier.sallenave@cortus.com, or visit http://www.lirmm.fr/~sallenave.

Roland Ducournau is Professor of Computer Science at the University of Montpellier. In the late 80s, while with Sema Group, he designed and developed the YAFOOL language, based on frames and prototypes and dedicated to knowledge based systems. His research topic focuses on class specialization and inheritance, especially multiple inheritance. His recent work is dedicated to the design and assessment of scalable constant-time techniques for implementing object-oriented languages. Contact him at `roland.ducournau@lirmm.fr`, or visit `http://www.lirmm.fr/~ducour`.

Acknowledgments We would like to thank: David Kerr-Munslow for his careful reviews, Floréal Morandat, who helped to put together the first version of this paper (thanks to his LaTeX skills), and the reviewers for their time and suggestions. This material is based upon work partially supported by the Association Nationale de la Recherche et de la Technologie.