Combining Generics, Pre-compilation and Sharing Between Software-Based Processes

Andrew Kennedy
Microsoft Research
Cambridge, U.K.
akenn@microsoft.com

Don Syme
Microsoft Research
Cambridge, U.K.
dsyme@microsoft.com

ABSTRACT

We describe problems that have arisen when combining the proposed design for *generics* for the Microsoft .NET Common Language Runtime (CLR) with two resource-related features supported by the Microsoft CLR implementation: *application domains* and *pre-compilation*. *Application domains* are "software based processes" and the interaction between application domains and generics stems from the fact that code and descriptors are generated on a *per-generic-instantiation* basis, and thus instantiations consume resources which are preferably both shareable and recoverable. *Pre-compilation* runs at install-time to reduce startup overheads. This interacts with application domain unloading: compilation units may contain shareable generated instantiations. The paper describes these interactions and the different approaches that can be used to avoid or ameliorate the problems.

1. INTRODUCTION

Parametric polymorphism, also known as "generics", is an important new feature of the C♯ programming language [12, 6] and also the .NET Common Language Runtime (CLR) [7] that underpins C♯ and other languages [9]. In previous work, the authors presented the design and implementation of generics for C♯ and the .NET CLR [10] — in this paper we call this design *Generic IL*. The primary novelty of the design is the integration of parameterized types and polymorphic methods into the type system of the intermediate language (IL) implemented by a virtual machine, or runtime. The implementation techniques described in [10] are novel in many ways: they include "just-in-time" specialization of classes and code, code-sharing between distinct instantiations of generic classes, and the efficient implementation of runtime types using dictionaries of type representations and computations.

This paper describes the interaction between Generic IL and two resource-related features supported by the Microsoft CLR implementation: *code sharing between software-isolated processes* (known as *application domains*) and *pre-compilation*.

In this section we describe Generic IL, application domains and pre-compilation, both by reference to the way these features appear in the prototype CLR implementation performed by the authors and also by describing the underlying assumptions that are relevant to this paper.

1.1 Generic IL

Generic IL is a set of extensions to the instruction set, metadata format and semantic rules for the Common IL of the .NET Framework. In this paper we ignore most of the design details of Generic IL and consider only the following aspects of the system:

- We assume a high-level language ultimately executed using native code (e.g. a high-level IL compiled using JIT compilation).
- We assume the language allows *named generic class definitions*, e.g. `List<T>`, and the instantiation of these types in client code.
- We assume the language supports *exact runtime type semantics*, i.e. casts and instance-of tests where casting to `List<string>` gives accurate results.
- We assume the language permits *instantiations at non-reference types*, i.e. e.g. `List<int>`, with the expectation that using these types gives "natural" performance, i.e. that faster code is generated for these types, and that expensive box/unbox operations are not typically needed.
- We consider cases where execution can require the *dynamic loading* of components, where new components may declare new instantiations of generic code.
- We assume IL code is arranged into components called *assemblies*, which are the unit of software installation and versioning. In this paper we use the words assembly and component interchangeably.

It is possible to implement Generic IL in a number of fundamentally different ways, and the choice affects the severity of the interactions described in this paper. However the interactions never disappear altogether for any of the design choices. Here we recap the primary implementation techniques for Generic IL:

No Code Specialization (Uniform Representations) There are several variations on this technique, with

a common pattern that there is a 1:1 correspondence between generic code and native code, i.e. only one copy of code is required for a generic type, which is used by all instantiations. It is used in many implementations of parametric polymorphism, and also in GJ [5]. Under some variations of this scheme, all values which are statically of variable type are represented as heap-allocated values. The advantage of this technique is that only one copy of code need ever be generated, and this code can be associated with the component which declares the named generic type. However this design choice has major ramifications for performance, especially when generic collection classes are instantiated with unboxed types. Furthermore, although only one copy of code is needed, type descriptors may still be required for each generic instantiation, in order to implement exact runtime type semantics.

Full Code Specialization This technique generates a new copy of code for each specialization. A preliminary performance analysis of the tradeoffs vis-a-vis full code sharing has been performed by Odersky et al. [15]

Mixed Code Specialization/Sharing Mixed code sharing/specialization places instantiations in equivalence classes of *compatible* representations.[1] For example, two types may be considered compatible if they are identical after erasing all reference types to `ref` ("reference type"), e.g. `List<string>` and `List<Widget>` are compatible if both `string` and `Widget` are reference types. Here `ref` is known as a *representation type*. Similar techniques have been applied in other contexts [3, 19, 11].

The prototype CLR implementation modified by the authors uses mixed code specialization/sharing. For most of this paper we simply assume that some kind of code specialization is being used, however in Section 5 we consider why code sharing helps to reduce the severity of the issues described in this paper.

1.2 Application Domains

Application domains are the CLR's notion of "in-process process" and can be unloaded, reclaiming the memory used by code and other VM data structures (which we will collectively refer to as simply "resources").[2] Isolation between application domains is ensured by type safety, and in many situations communication between application domains (e.g. remoted calls) can be performed more efficiently than between traditional operating system processes. They give large performance gains when used correctly, e.g. when used as part of a long-running server process, or when used as a temporary site of computation for code generated on the fly (complex regular expressions are evaluated in this way on the CLR). Typically objects may not migrate between domains, but application domains do share a common garbage collected heap. Application domains are also related to the management of code-access security permissions, and they are widely used in server-side programming.

Software components (*assemblies*) can be loaded as "shared" (also called *domain neutral*) and the native code for these components is shared between application domains. Shared components must form a strongly connected graph. Different application domains may have different sharing policies, an additional complexity not addressed in this paper.

Each application domain has its own copy of each static field — this is required to ensure domain isolation. This means that when shared code executes an instruction such as a "load-static-field" the results depend on the current application domain, and a *domain local store* (DLS) operation is needed, typically implemented as a table lookup based on an index stored in data private to the currently executing thread. Thus shared code incurs a performance penalty, with the aim that this is not significant in comparison to the savings achieved by reduced memory consumption in multi-application-domain scenarios. In the prototype CLR implementation modified by the authors these DLS operations are highly optimized but are still slower than the corresponding non-DLS actions — for example a DLS lookup for a static field might take 12 processor clock cycles, where the corresponding direct lookup would take 1.[3] While few computations are completely dominated by such accesses, the costs may still be incurred on dominant paths such as main loops.

There is a large body of work on supporting features similar to application domains in the context of other virtual machine designs, most recently those related to Java, but also many older projects such as SPIN (Modula-3) [4]. An excellent summary is provided in [2], where a huge range of systems is described, and in particular Alta, K0 and the J-Kernel [18]. Some entire operating systems are based on virtual machines [8]. All of these designs typically combine a process model with protection, resource management and communication. On some occasions these designs aim for more stringent accounting of resource usage than is provided by application domains, e.g. hard limits can be placed on the memory usage of software processes [1], or a GC may be modified to track and limit the dynamic memory used by an application [16].

1.3 Pre-compilation

Pre-compilation is a compilation operation typically run at install-time and was designed originally to reduce startup overheads related to JIT compilation. More recently, the increasing use of CLR-based code has seen an increase in the number of virtual machine processes that are likely to be running on any one machine, and pre-compilation is a convenient route to enable code and other pre-compiled VM data structures to be shared between multiple VM processes,

[1] This kind of code sharing is orthogonal to the main kind of resource sharing considered in this paper, i.e. resource sharing between application domains.

[2] "Other VM data structures" is used in this paper to cover the range of supporting vtables, caches, lookup tables and thunks used in a VM implementation, especially to implement features such as remote invocation and code access security. About all that one needs to know is the affinity of these structures to particular IL declarations, e.g. that these structures can be variously per-thread, per-method, per-instantiation, per-type-definition, per-assembly, per-domain or per-process.

[3] These "indicator" figures do not, of course, directly reflect the true costs of operations in modern memory hierarchies, e.g. an increase in pipeline stalls and/or cache misses.

since pre-compilation produces shareable DLLs.[4]

Pre-compilation is an emerging area of virtual machine implementation, and the architectural choices involved supporting a mixture of pre-compiled and dynamically-generated code have only recently begun to be explored, for example see the "quasi-static compilation" supported by the QuickSilver system [17]. (The pre-compilation techniques implemented by the Microsoft CLR implementation deserve study in their own right, partly because of the install-time nature of the service, and partly because of the care that has to be taken with regard to the interactions with versioning, security and other aspects of the VM design.)

In this paper we assume a model of pre-compilation where each component installed on the machine is pre-compiled at install-time.[5] Furthermore (and this is significant) we assume that the pre-compilation of each component must produce exactly one native binary that is suitable for management by the OS (e.g. a DLL on a Windows platform).

1.4 Aims of this paper and some additional assumptions

This paper describes problems that arise when you must support generics, application domains and pre-compilation within a virtual machine, under various assumptions about each of these features. We also make some additional assumptions about the nature of the VM implementation in question. These assumptions are true of the prototype CLR implementation and are typical of other implementations of OO languages. (We discuss the ramifications of these assumptions in Section 5):

Unique type descriptors Implementations of OO languages typically feature both *compiled code* and *type descriptors*. We assume type descriptors can be compared for equality by pointer comparison. In particular, we assume descriptors for constructed types such as `int[]` or `List<int>` need to be "hash-consed" even if they appear in multiple dynamically loaded assemblies. We assume this property need only hold true within each application domain, i.e. that different application domains may use different type handles for the same constructed type.

Type descriptors are Method Tables. We assume that type descriptors point to code, e.g. that type descriptors are traditional vtable-like structures. This is relevant because it means that type descriptors cannot outlive the code they point to.

No GC of code or descriptors. Application domains address the problem of resource reclamation, and garbage collection techniques an appealing alternative approach to this kind of problem. However, just as processes in an OS do not require sophisticated GC techniques, so application domains can be implemented without applying GC techniques to code or descriptors: unloading an application domain simply involves freeing all the data structures used by that domain. Furthermore shared components can also be unloaded when all application domains that reference those components have been unloaded — this only requires simple reference counting on shared assemblies. This is a relatively simple structural approach, and the aim of this paper is to investigate the choices when combining this simple approach with constructed types.

There are a range of solutions to the problems described here, with various tradeoffs with regard to throughput, memory usage, completeness and complexity. Additional possibilities arise if one is willing to change some of the assumptions indicated in the preceding sections. In order to keep things tractable we have concentrated on the techniques we have actively explored while implementing these features with a prototype Common Language Runtime.

1.5 An example of the problems that can occur

We now use a small example to illustrate some of the problems that can occur when combining the features described so far. We use C♯ syntax. First, let us define type definitions and some code fragments that occur in three shared components:

```
Shareable Assembly 0 (SA0)
   class List<T>
   {
      T[] elems;
      void List(T[] _elems) { elems = _elems; }
   }

Shareable Assembly 1 (SA1)
   class SortAlgorithms1
   {
      static void Sort1(List<int> x)  ...
      static void Sort2(List<string> x)  ...
   }

Shareable Assembly 2 (SA2)
   class SortAlgorithms2
   {
      static void Sort(List<int> x)
       { ... SortAlgorithms1.Sort1(x) ... }
   }
```

We assume execution involves generating native code for each instantiation of the `List` constructor, and generating unique type descriptors for `List<int>`, `int[]`, `List<string>` and `string[]`. Next assume we have two applications which are loaded as different application domains. The resources for these are not shared.

```
Application 1 (App1)
   class App1
   {
     static void Main()
     {
       int[] a = new int[] { 4,3 };
       List<int> b = new List<int>(a);
       SortAlgorithms1.Sort(b);
```

[4]This kind of inter-process sharing is distinct from the inter-app-domain/intra-process sharing referred to elsewhere in this paper

[5]It can be assumed that versioning is correctly managed, i.e. that all dependent assemblies are available at install-time, and if any updates to installed assemblies then any native images are invalidated and/or regenerated. Fortunately generics do not seem to create any extra difficulties with regard to the management of native image in the presence of versioning.

```
      string[] a2 = new string[] { "4","3" };
      List<string> b2 = new List<string>(a2);
      SortAlgorithms1.Sort2(b2);
    }
  }

Application 2 (App2)
  class Complex
  {
     Complex(float _x, float _y) { x = _x; y = _y; }
     float x;
     float y;
  }
  class App2
  {
    static void Main()
    {
      int[] a = new int[] { 4,3 };
      List<int> b = new List<int>[](a);
      SortAlgorithms2.Sort(b);
      Complex[] c = new Complex[]  ... ;
      List<Complex> a = new List<Complex>(c);
    }
  }
```

Importantly, we can ignore the details of the code and deal with a much simpler view of what the execution and/or compilation of the above code implies:

```
SA0:
  defines List<T>
  List<T> requires T[]

SA1:
  requires SA0
  requires List<int>
  requires List<string>

SA2:
  requires SA1
  requires List<int>

App1:
  requires SA1
  requires List<int>
  requires List<string>

App2:
  requires SA2
  requires List<int>
  requires List<Complex>
```

Now, assume we execute the following sequence:

```
Load Application 1
Load Application 2
Unload Application 1
Unload Application 2
```

We note:

- Loading `App1` creates type descriptors `List<int>`, `int[]`, `List<string>` and `string[]`. These are all used by the shared component `SA1`.
- Loading `App2` loads further shared code (`SA2`). This code uses `SA1` and we will assume that it expects to transact exactly the same type descriptors for `List<int>`, `int[]` as used by `SA1` and all other components. It also require descriptors for `List<Complex>` and `Complex[]`, however these won't be used by any other application domain, since the type `Complex` is an unshared type.
- When unloading `App1` we would like to release type descriptors `List<string>` and `string[]`. However, we cannot release the type descriptors `List<int>`, `int[]` since they are still in use by shared code running as part of `App2`.

Thus we have our first problem: **can we recover resources related to constructed types when application domains unload?**

Next, consider what happens if we run the sequence

```
Pre-compile SA0
Pre-compile SA1
Pre-compile SA2
Load Application 2
Load Application 1
Unload Application 2
Unload Application 1
```

Let us assume:

- Pre-compiling `SA0` does not create any instantiations.
- pre-compiling `SA1` places copies of the type descriptors `List<int>`, `int[]`, `List<string>` and `string[]` into the pre-compiled image for `SA1`.
- Pre-compiling `SA2` places copies of the type descriptors `List<int>`, `int[]` into the pre-compiled image for `SA2`.

Now for loading:

- We assume that when we load `App2` (which uses all three shared assemblies) the loader chooses to use the type descriptors `List<int>`, `int[]` from the pre-compiled image for `SA2` and `List<string>` and `string[]` from the pre-compiled image for `SA1`, i.e. that the loader chooses these as pointer-unique representatives for type descriptors.
- When `App1` is loaded, it uses shared code (`SA1`) also used by `App2` and we assume that shared code expects to transact the same type descriptors as chosen for `App2`.
- When unloading `App2` we would like to release the pre-compiled binary image for `SA2`. However, `App1` is still making use of resources from that image.

Thus we have our second problem: **under what circumstances can we utilize pre-compiled shared resources related to constructed types?**

Finally, consider what would happen if we placed a pre-compiled version of `List<int>` into the binary image for an *unshared* component, e.g. `App2`. This can lead to the third major problem: **can we utilize pre-compiled resources from unshared or unloadable pre-compiled binary images?**.

The manifestation of these problems obviously depends on a number of factors: uniqueness of type handles, for example, and in particular whether resources are unique across all

application domains. The remainder of this paper outlines various approaches to dealing with these problems. We do not give a quantitative analysis of the options in this paper, just a description of the problem and a qualitative analysis of the different characteristics of potential approaches.

2. COMBINING GENERICS AND APPLICATION DOMAINS

The interaction between application domains and generics stems from the fact that code and/or type descriptors are generated on a *per-instantiation* basis, and thus instantiations consume resources which are ideally both shareable and recoverable. These interactions exist for essentially all language implementations that combine *constructed types*, *exact runtime type semantics* and some kind of unloadable processes and would arise when combining a model of process in the context of Java-based systems.

2.1 Approach 1: Always share where possible

The simplest approach to dealing with constructed types in the presence of code-sharing for certain components is to *always share the resources associated with a constructed type if all the constituent components of the constructed type are shared.*

In particular, assume that generic instantiations are stored in a table (called the *instantiation-table*). In the absence of application domains, this table is global to an entire virtual machine process and resources associated with instantiations do not need to be collected. However in the presence of application domains we must decide the lifetimes for these resources. If the resources are to be shared we need an instantiation-table that is global across all applications domains. Resources related to unshared instantiations are stored in an additional table associated with the application domain. That is, data structures relating to generic instantiations must be placed in either the application domain or be "domain neutral".

An unsound approach is to share all resources associated with all constructed types. Consider what happens when a constructed type involves a type from an unshared component, e.g. the type `List<Complex>` from the example in Section 1.5. If the resources for such a type are shared then a leak and/or a dangling reference will occur when the unshared component is unloaded. Thus we define *potentially-shareable* and *never-shareable* according to the constituent components involed:

- A constructed type is *potentially-shareable* if all the constituent components of the type are defined in shared components. For example `List<int>` is potentially-shareable if both `List` and `int` are defined in shared components.
- A constructed type is *never-shareable* if one of the constituent components of the type is from an unshared component. For example `List<MyAppType>` is never-shareable if `MyAppType` is defined in an unshared component.

Consider the example from Section 1.5. The potentially-shareable constructed types required during execution are `List<int>`, `List<string>`, `int[]` and `string[]`. The constructed types `List<Complex>` and `Complex[]` are never-shareable.

A sound approach is to always share potentially-shareable instantiations, and indeed this is the basic approach taken in the prototype CLR implementation performed by the authors.

However this approach reduces the resource reclamation that can be performed when application domains are unloaded, at least if shareable items are not individually collectable, e.g. are placed in a non-collectable global table as indicated above. In this case the resources associated with a constructed type such as `List<int>` will never be collected, regardless of the application domains running. The resources may be of use to future application domains, and indeed a common usage pattern for application domains is to repeatedly load the same program. However, ideally the resources must be reclaimable when no longer in use.

A refinement to this scheme is to individually reference count the resources associated with instantiations, just as shared components are reference counted. However, this relies on the assumption that these resources are "individually collectable". We shall see that the individual collectability of items is much more difficult in the presence of precompilation.

2.2 Approach 2: Never share

A sound approach which permits reclaiming of resources is to never share resources associated with instantiations. Note that resources associated with simple named types can still be shared — we do not abandon resource sharing between application domains altogether. That is, each application domain would have its own copy of the code and descriptors related to each constructed type. This ensures that all resources used by an application domain are recovered, since the resources can simply be released when the application domain is unloaded. However this comes with a major cost: uses of these resources from shared code will require *domain local store* operations (see Section 1.2). Thus, although this is a structurally sound solution with regard to resource reclamation, it has problems:

- The extra overhead of DLS operations would appear high. For example, every operation such as `new List<int>` from within domain neutral (shared) code may incur an additional overhead.
- The approach means resources associated with constructed types are never shared between application domains.

2.2.1 Example

Consider the example from Section 1.5. We can now reanalyze the behaviour for the following sequence:

```
Load Application 1
Load Application 2
Unload Application 1
Unload Application 2
```

We note:

- Loading `App1` will create type descriptors `List<int>`, `int[]`, `List<string>` and `string[]`. These are all used by the shared component `SA1`. However, no direct references to these type descriptors or their associated code will be baked into the code for `SA1` — instead DLS operations will be used whenever the descriptors

are required. For example, assume the code `Sort1` is a merge-sort algorithm that creates new lists

```
static void Sort1(List<int> x) {
  ... Sort1(new List<int>  ...  ) ...
}
```

The allocation operation will create an object and will need to tag this object with the exact type `List<int>`. This type descriptor will be found at runtime by a DLS lookup.

- Loading `App2` will load further shared code (`SA2`). This code uses `SA1`, but unlike the analysis in Section 1.5 we no longer need to expect to transact identical type descriptors for `List<int>`, `int[]` to those used by `App1`. Thus we can allocate new descriptors for use by `App2`. As mentioned in Section 1.5 it will also require descriptors for `List<Complex>` and `Complex[]`, however these are always specific to the application domain, since the type `Complex` is an unshared type.

- Now, when unloading `App1` we can simply release all type descriptors which are relative to that application domain.

Thus this approach solves the first problem mentioned in Section 1.5: we recover resources related to constructed types when application domains unload.

2.2.2 *Refinement: Speculative sharing*

A useful refinement to this approach may be to speculatively choose to share some potentially-shareable instantiations. In other words we make a global choice and divide potentially-shareable instantiations into two sets: instantiations always to be shared, instantiations never to be shared. For example, if an instantiation is referenced by a core system library which is never to be unloaded, then we can share it. Similarly, if a type `List<int>` is required by the component which defines either the type `List` or the type `int` then we know the resources will be required for as long as the constituent types are required, and we can choose to share the type safely.

We can also speculatively place items in the shareable set, on the assumption that even if no application domains use the item then the items are still likely to be used by future application domains. For example, a common type such as `List<int>` might be treated in this way. If this choice is made sufficiently globally then all DLS overheads associated with looking up this type can be completely eliminated.

2.2.3 *Refinement: Partial sharing*

Finally, one can also modify the never-share technique to share instantiations amongst a subset of application domains. For example, an application domain can "publish" an item and associate that item with one of its shared components. Other application domains may then use that item if (and only if) they also make use of that shared component. This enables some sharing but does not remove the overheads of DLS operations. We will return to this approach in Section 4.

3. PRE-COMPILATION AND GENERICS

The remainder of this paper deals with complications that arise when we combine pre-compilation with the approaches described in Section 2 to tackle the combination of generics and application domains. However, first we must describe the approach taken to pre-compilation for generics.

Generics present problems for pre-compilation, precisely because the implementation techniques described in [10] rely on global tables of available instantiations being managed by the VM. Here we give an overview of the techniques used by the prototype implementation of GenericIL in the Microsoft CLR. However, the exact details of the technique are not essential to this paper — what matters is only the following fact: *some code and/or descriptors are pre-compiled into native images for client assemblies, rather than into the native image for the component where the generic code is defined.*

There are several approaches possible when combining pre-compilation with generics. Let us assume that we have a generic type `ClassFromAssembly1<T>` and several uses of this type at instantiations which require different code:

No pre-compilation of instantiations. Don't pre-compile any instantiations of generic types, relying on dynamic JIT specialization and dynamic type descriptor creation to create all instantiations as necessary.

Pre-compilation into the declaring component. Selectively pre-compile the code for a small number of representative instantiations, placing the results of the compilation into the native image for the component that declares the generic type. For example, we may selectively decide to pre-compile the code for the type representation `ClassFromAssembly1<ref>` into `Assembly1`.

Pre-compilation into client assemblies. Under this technique, if a component is statically determined to potentially require an instantiation, then a copy of the instantiation is compiled into the native image for that component. For example, if `Assembly3` uses instantiation `ClassFromAssembly1<ClassFromAssembly2>` then a copy of that instantiation will be compiled into the native image for `Assembly3`. An important optimization is to avoid duplicating any instantiations which have already been compiled into referenced assemblies. This process can lead to multiple copies of code being compiled when two or more assemblies independently reference an instantiation. We assume the dynamic loader of the virtual machine ensures that only one such instantiation is used at run-time.

The prototype CLR implementation uses a mixture of these techniques: "pre-compilation into the declaring component" of the code for the `<ref>` instantiation, combined with "as-required pre-compilation into client assemblies" for some other instantiations. Similar techniques are used for descriptors. The aim is to achieve a *pre-compilation guarantee* that if particular rules are followed then no runtime compilation of code will occur.

Pre-compilation may be applied both to shared and unshared components. The pre-compiled image for an unshared component can be used in at most one application domain at any one time.

4. COMBINING GENERICS, APPLICATION DOMAINS AND PRE-COMPILATION

We now come to the final set of interactions studied in this paper: those that arise when supporting generics, application domains and pre-compilation in combination. The primary additional difficulty caused by pre-compilation with regard to resource usage and reclamation is that it effectively "binds together" a set of pre-generated resources into a single unit. That is, if any resources in a pre-compiled binary image are live then the image itself may not be released, e.g. its file descriptor may not be closed and/or the image may not unmapped from memory. In effect, using one resource in a pre-compiled component means using all resources in that component.[6]

For a named type (let's say `TypeInAssembly1`) this is not a significant problem, because resources associated with this type can be accounted to the originating component (in this case `Assembly1`). When the component is no longer referenced by any application domains the whole image can be released. Thus resource usage can be managed at the granularity of assemblies.

However for constructed types problems arise. When the resources for an instantiation (say `List<int>`) are placed into a pre-compiled image for a component (say a client component otherwise unrelated to `List` or `int`) then the resources associated with this type effectively become associated with the client component. These problems manifest themselves in a number of concrete ways, which we now explore in more depth. These problems only arise when the "always-share" approach from Section 2.1 is used as the technique for managing the interactions between generics and application domains.

4.1 Unshared images containing shared resources

There are situations where the approach to pre-compilation of generics outlined in Section 3 means that pre-compiled unshared components would contain resources for potentially-shareable instantiations. For example, if a unshared component `A` refers to a potentially-shareable type `List<int>` then we would like to generate the resources for `List<int>` in the pre-compiled binary for `A` (we would only do this if no pre-compilation of `List<int>` is available in other pre-compiled images accessible to `A`). However, if `A` is an unshared component then the usefulness of these pre-compiled resources is limited.

First, resources related to potentially-shareable types but which are located in an unshared image are of no use to other application domains: the image is an unshared image, and no resources from it may be used by other application domains.

Furthermore, the resources are not necessarily of use within the context of the application domain itself. This depends on the approach to combining application domains and generics, outlined in Section 2:

- If the "always-share" approach from Section 2.1 is used then descriptors must be unique across multiple application domains. However, as mentioned above descriptors from an unshared pre-compiled binary are not suitable for use by other application domains.
- If the "never-share" approach from Section 2.2 is used then no further problems arise: the resources may be used for the application domain itself. The "partial-sharing" approach from Section 2.2.3 will also work correctly.

Thus, if the always-share approach is used for combining constructed types with application domains, then only limited pre-compilation guarantees are provided by the approach to pre-compilation outlined in Section 3. Not all instantiations placed in pre-compiled images can be used at runtime, because shared instantiations in unshared images are of no use. No actual unsoundness arises: it is just that some instantiations must be dynamically created/compiled at runtime.

4.2 Shared images containing shared resources

Like unshared components, the pre-compilation for shared components can involve including resources associated with constructed types in the pre-compiled images. For example, if a shared component `A` refers to a potentially-shareable type `List<int>` then we may generate the resources for `List<int>` in the pre-compiled binary image for `A`.

If shared components are never unloaded, then these pre-compiled resources are always usable (subject to the constraint that we only make use of at most one copy of a type descriptor for any particular instantiation). In this situation the resources have the same properties as the shared resources created dynamically for shareable types. This applies regardless of which of the techniques from Sections 2.1-2.2.3 is used.

The prototype CLR implementation performed by the authors did not support the unloading of shared components. However, the ideal is that shared components should be unloadable once they are no longer referenced by any application domains. If we naively attempt to reach this ideal by simply unloading shared native images then similar problems arise to those described above for unshared components: we must restrict the use of pre-compiled generic instantiations from native images, and even weaker pre-compilation guarantees result.

For example, when using the "always-share" approach, if we do not take care then pre-compiled shared components may contribute resources which are then utilized by other, otherwise unrelated shared components. An example of this was seen in Section 1.5. This would in turn cause an unsoundness when the shared components are unloaded. Furthermore, the refinements mentioned in Section 2.1 to address the problems of resource reclamation when using the "always-share" approach assumed that resources associated with constructed types could be collected individually. However, pre-compilation breaks this assumption.

Thus, the primary effect of combining the "always-share" approach with pre-compilation is that we get weaker pre-compilation guarantees than one would wish for.

5. DISCUSSION AND RELATED WORK

[6]Binary image formats are, of course, more flexible than this, and may contain multiple sections which can be loaded separately. However, the resources within each section are still related, and furthermore the image file itself will still be in use if any of the sections are in use.

This paper has described issues that have arisen when prototyping the proposed design for Generic IL with the application domain and pre-compilation features in a prototype CLR implementation. The main purpose of the paper has been to highlight the existence of these interactions and to describe the implementation options in a qualitative fashion: if you're going to implement a version of the CLR with generics and/or pre-compilation then it will be important to address these issues! The interactions were somewhat unexpected: generics is a language feature and there is little overlap between OS research and this class of language features. However the interactions are unsurprising in retrospect: generics consume resources, and resource management is not always straight-forward. The issues stem from the fact that code and descriptors are generated on a *per-instantiation* basis, and thus instantiations consume resources which are preferably both shareable and recoverable. Secondly, these resources cannot be accounted for in a way that fits nicely with existing resource-accounting and reclamation mechanisms inside typical CLR implementations.

The issues described arise primarily when using the "always-share" approach from Section 2.1. The alternatives we have described look plausible from a design perspective but need careful analysis with regard to the tradeoffs between pre-compilation, throughput and memory usage. We sketch an additional alternative below. Alternatives such as "never-share" and its refinements would incur extra overheads associated with domain-local-store lookup operations, which need to be closely assessed.

The problems described arose in practice in the prototype CLR implementation performed by the authors. In practice, the primary outward effects of these problems were that the pre-compilation guarantees provided by that system were slightly weaker than one would wish: unshared components do not necessarily have all instantiations available as pre-compiled. In particular, shared resources related to constructed types were not pre-compiled, unless they were pre-compiled into some shared image. A secondary effect was that resources associated with shared instantiations were not reclaimed when the application domains that use these instantiations are unloaded. To some extent, both effects were already present in the original CLR implementation on which the prototype was based, because that prototype supported array types, which also consumed resources on a per-type basis.

As mentioned in the introduction, our prototype CLR implementation uses shared code between different instantiations of generic code (this is orthogonal to code sharing between application domains). This greatly reduces the manifestation of the above issues: in particular, we pre-compiled the native code for "reference" instantiations into the image for the assembly declaring the generic code. This means the above issues do not appear for the code associated with reference instantiations, and also means that this code is never duplicated. By further increasing the level of code sharing (and in general reducing reliance on per-instantiation resources) it is possible to further improve the properties of the system.

5.1 Future Work: Non-uniqueness

In Section 1 we made the assumption that type descriptors and code must be unique within each application domain, i.e. equality comparisons can be implemented as pointer equality. Although a full treatment is beyond the scope of this paper, we mention here that it appears possible that one can build alternative solutions by weakening this assumption. The primary downsides are (a) a potential for more expensive exact-runtime type tests and (b) greater memory usage. The second of these effects can be offset by the fact that we no longer need to store hash-consing tables to uniquify items, and by a reduction in the "uniquifying fixups" that need to be applied to pre-compiled code.

However, several issues remain. If resources related to constructed types need no longer be unique, then how many copies of resources do we create, and which copy of a descriptor will each component reference? In particular, how many copies do we need if we are to support full resource reclamation, strong pre-compilation guarantees and minimal DLS-related overheads? The best approach to use would appear to be to create one copy of each resource for each component that directly references a resource, with an optimization that we can reuse resources from components transitively referenced by a shared component: this is similar to the approach described for pre-compilation in Section 3, except it would apply to JIT compilation/loading as well.

It is also interesting to consider other language features and that may relate to the interactions described in this paper. For example, even some simple language features such as exceptions in Standard ML are "generative" to a minor degree, i.e. they consume resources (such as allocating an integer tag) as additional code is loaded.

5.2 Future Work: Quantitative analysis

In this paper we have deliberateily used qualitative techniques to describe the choices and tradeoffs in the internal architectural design for a virtual machine that supports a combination of generics, application domains and pre-compilation. Naturally, a thorough quantitative analysis of the choices is also highly desireable, but was beyond the scope of this paper.

5.3 Related Work

As discussed in the introduction there is a very large body of work looking at the boundary between language-based extensible systems and operating systems — see [2] for an excellent overview. The issues discussed in this paper would arise when a model of generics with per-instantiation resource usage is combined with almost any of the systems described there, and likewise with the addition of both generics and pre-compilation.

The authors are not aware of work that focuses explicitly on resource issues associated with constructed types, or issues related to pre-compilation. To some extent this is surprising, since some of the interactions described here also apply to array types. However, systems such as Java typically have a model of array types which is not as rich as the CLR's — for example in Java there are a limited number of reference array types, and multi-dimensional array types are always "jagged". This means it is normal to implement JVM array types without using resources for each individual array type.

Some of the problems described in this paper are similar to the problem of interning strings across software-based processes, to the extent that unique pointers are assumed for type handles. For example, the KaffeOS [1] places a language limitation on string interning, refusing to intern

strings across software processes, a feature which may be semantically visible in some situations, though only when using KaffeOS primitives to communicate or share between processes. The tradeoffs for string interning are similar to those described in Section 2.

Surprisingly, the authors have not been able to find work focused on the problem of supporting C++ template-like features in the context of dynamic linking and separate compilation, which would also pose a very similar set of problems to those described here.

5.4 Related Material

The source code for an early version of the prototype implementation referred to in this paper has been made available as a modification of the "Rotor" CLR implementation [14, 13].

5.5 Acknowledgements

We would like to thank Sean Trowbridge, Vance Morrison, Simon Peyton-Jones, Chris Brumme, Richard Black and Claudio Russo for helpful discussions related to this work. We also thanks the referees for their helpful comments and observations.

6. REFERENCES

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000. USENIX.

[2] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the design of Java operating systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 197–210, June 2000.

[3] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[4] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Egger. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating Systems Principles.*, pages 267–283, 1995.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[6] ECMA International. ECMA Standard 334: C# language specification. Available at `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[7] ECMA International. ECMA Standard 335: Common Language Infrastructure. Available at `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[8] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The JX operating system. In *USENIX Annual Technical Conference*, pages 45–58, June 2002.

[9] A. Hejlsberg. The C♯ programming language. Invited talk at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2002.

[10] A. J. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*. ACM, 2001.

[11] X. Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation, Amsterdam*, June 1997.

[12] Microsoft Corporation. An introduction to C# generics. See website at `http://msdn.microsoft.com/vcsharp`.

[13] Microsoft Corporation. The Microsoft Shared Source Common Language Infrastructure (Rotor). See website at `http://msdn.microsoft.com/net/sscli`.

[14] Microsoft Research, Cambridge. Gyro: Generics for the SSCLI. See website at `http://research.microsoft.com/projects/clrgen`.

[15] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your pizza — translating parameterised types into Java. In *Generic Programming*, pages 114–132, 1998.

[16] D. Price, A. Rudys, and D. Wallach. Garbage collector memory accounting in language-based systems. Technical Report TR02-407, Dept. of Computer Science, Rice University, November 2002.

[17] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Conference on Object-Oriented*, pages 66–82, 2000.

[18] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, 1998.

[19] M. Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Selected Areas in Cryptography*, pages 610–619, 2001.