Pre-compilation for .NET Generics

Andrew Kennedy
Don Syme

Microsoft Research, Cambridge, U.K.

Abstract. The Microsoft .NET Common Language Runtime (CLR) supports *pre-compilation* in order to help avoid the slow application start-up times and unnecessarily high memory usage often associated with virtual machine (VM) execution, However, Version 2 of the .NET Common IL (CIL) also supports *generics*, i.e. type parameters for data and code, in part implemented using template-expansion techniques, and the use of dynamic compilation for generics would threaten the gains of pre-compilation. We describes techniques to support the pre-compilation of generic CIL code, resulting in the first language implementations to support the combination of dynamic loading, pre-compilation and type-parameterization over unboxed representations. We discuss the load-time techniques used to resolve potential duplication and optimizations based on "preferred pre-compilation modules". We give performance results relating to code-size, code-duplication and start-up times as observed in our implementation of the techniques, aspects of which have been shipped in recent releases of v2.0 the Microsoft .NET CLR.

1 Introduction

Virtual machines (VMs) that perform Just-In-Time (JIT) link-loading and compilation can exhibit high runtime compilation costs, leading to slow application start-up. For example, a widely used implementation of the Java Virtual Machine$^{\text{TM}}$(JVM) has been reported to have start-up times for "Hello World" in the order of 100 times slower than the equivalent Perl program [1]. To help avoid similar problems the Microsoft .NET Common Language Runtime (CLR) supports *pre-compilation*, whereby some or all of the costs of IL-to-native translation is amortized, e.g. at install-time. Pre-compilation is essentially traditional compilation applied to Common IL (CIL), leaving a residue of link-load fix-ups, and, if necessary, further compilation can be performed at runtime in order to support the dynamic tuning of code. For many purposes pre-compilation gives start-up times in the order of corresponding native executables. Furthermore, pre-compilation is applied to most foundational libraries by default, so library-dominated programs start efficiently without explicitly applying pre-compilation. Pre-compilation as implemented by .NET has proved decisively that, given the right architectural choices, start-up times need not be an issue for most programs executing on VMs.

In addition, VMs often fail to share code and other compilands across multiple hardware-isolated OS processes. To quote from a recent paper on code sharing in JVMs [6]:

> The idea of sharing executable code gained widespread acceptance in the mid-1980s...Shared libraries lower the system-wide memory footprint... [It] is common to come across a computer running many applications written in the Java™ programming language at any given time. One might ask whether in these settings the sharing of executable code across multiple virtual machines is as beneficial for the scalability of the JVM as shared libraries are for OSs.

The Microsoft .NET CLR implementation achieves sharing of compilands simply by ensuring that pre-compilation itself produces shared libraries. That is, once pre-compilation is completed, sharing is automatic through the well-known, reliable and efficient techniques supported by most operating systems. This is in contrast to proposed techniques for JVM, which require either sophisticated "dynamic" sharing techniques [6] or a single virtual machine process, both leading to problems with security and robustness.

However, pre-compilation interacts heavily with features that rely on dynamic compilation. In particular, Version 2 of CIL includes support for *type parameters*, which provide the execution foundation for *generics* as a feature of C♯, VisualBasic.NET, F♯ and other programming languages [12, 7]. In previous work, the authors presented the design and implementation of generics for C♯ and the .NET CLR [8] — in this paper we call this design *Generic IL*. The primary novelty of Generic IL is the integration of parameterized types and polymorphic methods into the type system of the IL implemented by a VM.

Some of the benefits of Generic IL arise because it is typically implemented using mixed code expansion and sharing, in contrast to generics for the Java™ language which are implemented by erasure in the source language compiler [4]. Furthermore the exact runtime-type semantics of Generic IL requires that dictionaries and descriptors be generated as new instantiations arise. However together these mean that in the worst case we must assume that a JIT-compiler is available to dynamically generate instantiations, and that the link/load tables of a VM can be used to manage generation and sharing of instantiations.

This paper describes how to support pre-compilation for nearly all Generic IL code. The contributions of this paper are as follows:

- We describe the techniques used to achieve the first implementation of separate compilation for Generic IL, thus giving the first dynamically-loading language implementations where generic constructs are implemented using template-expansion techniques;[1]
- We describe a novel set of optimizations called *stable homes* and *preferred homes* for compilands related to Generic IL instantiations;
- We give a series of performance results to confirm the importance of supporting pre-compilation for Generic IL;

[1] Note that code using C++ templates may be dynamically loaded, but templates may not be used across binary boundaries.

– We give background material on .NET pre-compilation, whose existence has surprisingly been ignored by recent published work in this area.

The approach we describe achieves the twin goals of eliminating the start-up costs and sharing resources across multiple VM instances, and we give performance results as evidence of this. The implementation strategy is based on

1. a degree of duplication of code into multiple pre-compilation binaries;
2. the optimizations mentioned above;
3. the use of JIT compilation for some remaining corner cases related to polymorphic recursion and generic virtual methods.

We also discuss other possible approaches, though the complexities of implementing various alternatives has meant that we have only been able to give quantitative analysis for the techniques actively explored while implementing these features within the Microsoft .NET CLR implementation.

2 Background

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

2.1 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 consider only the following aspects of the system:

– We assume a high-level language ultimately executed using native code.
– 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*, e.g. casts and instance-of tests where casting to `List<string>` gives accurate results.
– We assume the language permits *instantiations at non-reference types*, 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, here 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 possible implementation techniques for Generic IL:

No Code Specialization (Uniform Representations) All values that are statically of variable type are represented as boxed, heap-allocated values, a technique used in many implementations of parametric polymorphism. Thus gives is a 1:1 correspondence between generic code and native code, but has major ramifications for performance when generic collection classes are instantiated with unboxed types. For Generic IL type descriptors would still be required to implement exact runtime type semantics.

Full Code Specialization With this technique, a new copy of code is generated for each specialization, a technique used in many implementations of C++ templates.

Mixed Code Specialization/Sharing Mixed code sharing/specialization places instantiations in equivalence classes of *compatible* representations. 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*.

The prototype CLR implementation modified by the authors uses mixed code specialization/sharing. We will assume the existence of a family of *representation types* and a function `rep`(T) that generates the representation type for each Common IL type, and indeed assume that this representation function is precisely the "reference-type-erasure" function described above.

2.2 Pre-compilation

The rationale for pre-compilation has been explored in the introduction. In this section we provide additional background information relevant to the remainder of this paper.

Below is an example of using `ngen` tool to pre-compile an application for the Microsoft .NET Framework 2.0. The left shows start-up performance of "Hello World" with all components pre-compiled, the middle with only the system libraries pre-compiled, and the right with no components pre-compiled.

```
> ngen install mscorlib   > ngen delete *           > ngen delete *
> ngen install hello.exe  > ngen install mscorlib
> utime -u hello.exe      > utime -u hello.exe      > utime -u hello.exe
Hello World               Hello World               Hello World
user: 0:00:00.031         user: 0:00:00.031         user: 0:00:00.312
```

In theory, pre-compilation involves a spectrum of possible choices: a compilation architecture could choose to pre-compile none, some or all of the code in an

application. For example, we later show performance figures with only the non-generic code in an application pre-compiled. Similarly data structures such as static garbage collection tables can be either saved or JIT-computed, and these decisions can even be made based on profiling feedback.

Pre-compilation is not a guaranteed performance gain: at the extreme it may be optimal to pre-compile nothing at all. Although pre-compilation usually give significant overall performance gains, a number of factors affect the overall equation:

- Pre-compilation may interact with the memory hierarchy, e.g. pre-compiled code is typically on mapped memory pages that cannot easily be subjected to optimizations such as compaction or fine-granularity garbage-collection.
- In theory, pre-compilation provides an opportunity to apply more extensive optimizations than are normally performed by JIT compilers. We do not make use of this potential in the implementations analyzed in this release.
- Pre-compiled code requires indirections, link-time fixups and linkage information, as references to items in other assemblies may need indirection slots and/or linkage "fixups" (equivalent to symbol names in traditional dynamically loaded images).
- Pre-compiled code prefers read-only code and data structures. Data structures recorded in pre-compiled images may be further modified during execution, but this is implemented using the copy-on-write memory paging primitives provided by most operating systems. Thus each modified page becomes "private" to a hardware-isolated process. It is always better to eliminate private pages in favour of shared pages, e.g. by making pre-compiled data structures and code read-only.

In short, the performance trade-offs for pre-compiled code are non-trivial, but the overwhelming motivation is to reduce the start-up times of applications.

2.3 The Granularity of Pre-compilation

This paper only considers a model of pre-compilation where each source component (actually an *assembly* in the context of Common IL) is compiled to a single native code binary image - thus we can speak of the *pre-compiled image for an assembly*. .NET assemblies tend to be fairly coarse - of the order of 100kb or more of .NET IL - and indeed the use of assemblies as the granularity of component distribution and installation is one of the architectural choices in .NET that appears to have helped to enable efficient pre-compilation. Attempts to share compiled code in the context of JVM implementations have encountered the problem that the granularity of sharing is too small [6].

As such we do not consider architectures based on smaller granularities, nor on on-disk or shared-in-memory caches of instantiations. We also assume the granularity of assemblies is relatively coarse, i.e. that assemblies often contain hundreds or thousands of classes, rather than just a small number. This means considerable internal sharing can be achieved during the pre-compilation of each assembly.

2.4 The Order of Pre-compilation

Common IL assemblies declare their *static dependencies* on other assemblies, i.e. a set of assembly references resolved by the virtual-machine at pre-compilation and load-time. At load-time the resolution is checked to ensure it matches the resolution performed at pre-compilation-time: if any mismatches are found the pre-compiled images cannot be used. 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. Generics are in many ways similar to inlined functions, and do not create any extra difficulties with regard to the management of these native image in the presence of versioning.

In some situations it can be important to manage the "fragility" of native images and re-compilation — for example, it may be desirable to produce images where one re-compilation does not induce the re-compilation of all dependent assemblies. The `ngen` tool pre-compiles all assemblies (e.g. shared libraries) which the installed application depends upon in a hard-bound fashion, unless they have already been compiled.

Pre-compilation normally processes each binary independently, i.e. no use may be made of the binaries that result from prior compilations. However, version 2.0 of Microsoft .NET CLR implementations permit Common IL assemblies to declare via attributes that they have a "hard-bound" dependency on other assemblies, in which case the latter are guaranteed to be pre-compiled before the former, and the results of compilation of the latter can be used when pre-compiling the former.

3 Interactions between Generics and Pre-compilation

As mentioned in the introduction, generics present problems for pre-compilation, precisely because the implementation techniques described in [8] rely on JIT-compilation and global tables of available instantiations managed by the VM. We now use some small examples to illustrate some of the issues that can occur with regard to generics and pre-compilation.

3.1 Instantiations have no natural "pre-compilation home"

The most fundamental problem can be characterized as follows: it is not possible to assign an appropriate "pre-compilation home" for each instantiation, i.e. a native image that we know *a priori* will be used to host the compilands (i.e. code and descriptors) for an instantiation. Diamond dependencies are the classic example of this problem:

```
Assembly 1.dll: Declares List<T>
Assembly 2.dll: Declares class C
Assembly 3.dll: Depends on 1,2, Instantiates List<C>
Assembly 4.dll: Depends on 1,2, Instantiates List<C>
Assembly 5.exe: Depends on 1,2,3,4
```

Clearly the instantiation `List<C>` is required by 3.dll and 4.dll but it would be incorrect, or at least non-optimal, to assign either as its home. This could be characterized by saying that in the presence of generics it is not possible to construct a mapping from `Home` : *type* → *assembly* such that each assembly that uses type τ has a static dependency on `Home`(τ). In the absence of constructed types this is possible, by simply using the assembly containing each named type as its "home". It is also possble in the presence of only a fixed set of pre-declared unary type constructors - for example the home for an array type can be declared to be the home of the element type.

3.2 The Problem of Multiple Type Descriptors

Implementations of OO languages typically feature both *compiled code* and *type descriptors*, and in general assume that type descriptors must be *pointer-unique*, i.e. 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 unique even if potential copies of these descriptors appear in multiple dynamically loaded assemblies.

If the same descriptors are placed in multiple pre-compiled native images then clearly either the logic that implements type equivalence must cope with multiple descriptors, or else only one descriptor may be "used" at runtime, with the consequence that even intra-assembly references to these descriptors may need to be subjected to indirections and link-time fixups. We consider this further in Section 4.6.

3.3 Polymorphic Recursion and Generic Virtual Methods

Two features of Generic IL pose particular problems for pre-compilation: *polymorphic recursion* and *generic virtual methods*.

Polymorphic recursion occurs when generic instantiations refer to more complex instantiations in an expansive fashion. Most examples are pathological, but are difficult to rule out statically in a system with dynamic loading. In the following example, `MyMethod` will return an object with runtime type `List`^{n}`<T>`.

```
static object MyMethod<T>(int n) {
  if (n > 1) return MyMethod<List<T>>(n-1);
  else return (object)(new List<T>()); }
```

Clearly, execution of such functions may involve a potentially unbounded number of instantiations, and thus pre-compilation has no hope of generating a suitable closure of instantiations.

Generic virtual methods, i.e. virtual methods that themselves take generic parameters, are effectively a form of "runtime type application". The authors have analyzed their expressive power via contrast with System F [10]. Like most indirect calls, the target of a generic virtual method cannot always be resolved at compile time, and in such cases neither a JIT nor pre-compiler can determine which target code to instantiate with the given instantiation. Thus it is not

possible to pre-compile all such instantiations, except via a global analysis that pessimistically instantiates all targets with all instantiations. In this paper we do not aim to achieve any pre-compilation guarantees for such cases.

3.4 Multiple Application Domains

Elsewhere the authors have described an interaction that exists between generics, pre-compilation and a feature of Common IL called *application domains*, which are a form of *software isolated processes* [9]. The issues involve resource reclamation in the presence of unloadable compilation units. Since this issue has been discussed in detail elsewhere we do not consider it further in this paper.

4 Supporting Pre-compilation for Generic IL

In this section we give an overview of the techniques used in our implementation of pre-compilation for the Microsoft .NET CLR.

4.1 An Overview of Possible Approaches

Within the overall constraints described in Section 1 we can proceed to tackle the problem of pre-compilation for generic code in several different ways, which we now illustrate. Let us assume that we have a generic type `ClassFromAssembly1<T>` and several uses of this type at different representative instantiations. Several strategies are possible:

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 clearly lead to multiple copies of code being compiled when two or more assemblies independently reference an instantiation.

Which approach is superior will depend on many factors, including the way in which generics are used by client code. The first option is appealing in many

ways, as it is conceptually simple and ensures that no duplication occurs. However, generics tend to be used extensively across core system libraries, shared user libraries and within client applications. Clearly it is desirable to be able to pre-compile uses of generics in all these situations. Furthermore, in some high-performance scenarios the ideal is that pre-compilation give a guarantee *no* JIT-compilation occurs except in very specific circumstance (e.g. when certain language features or libraries are used). In addition, the advantages that pre-compilation gives across multiple hardware-isolated processes means that the pure JIT-compilation of instantiations appears especially unsuitable for core libraries where a high degree of cross-hardware-process sharing is required.

4.2 Our Chosen Approach

Given the choices above, our starting point is as follows:

- The pre-compilation of each assembly is augmented to include some of the transitive closure of instantiations induced by ground instantiations in that assembly.
- The rules of the transitive closure are derived from the static requirements of the generic templates involved: if a generic class `Stack<T>` requires another generic class `List<T>` then a production `Stack<T> --> List<T>` is included in the rules for the transitive closure.
- We generate the transitive closure only with respect to the *type representation* function `rep`(T) described in Section 2.1. This means that the transitive closure is generated with respect to *representative* instantiations rather than exact instantiations, and new code will only be generated as new representative instantiations are detected.
- The pre-compilation is augmented by instantiating each generic class and method with its "reference instantiation" (`<ref,...,ref>`) instantiation, where `ref` is the type representation for all reference types. This is the code for the case where each type variable is instantiated to a reference type.
- We limit the transitive closure to include only those instantiations which have not already been generated in the pre-compiled images for hard-bound assembly dependencies (see Section 2.4).

The transitive closure described above can be computed at different granularities, for example the granularity of individual types (if one method is required in a type, then it is assumed that all methods are required), or at the granularity of individual methods (e.g. productions of the form `Stack<T>::Length() --> List<T>::Length()`). For the remainder of this paper we assume that granularity is tracked at the level of whole types, thus if `Stack<int>` is required at any point then all methods in `Stack<int>` will be pre-compiled, regardless of whether they are directly called.

Generic IL also includes instantiations of generic methods, and these must also be tracked, e.g. items such as `List<C>::Map<D>` may appear in the graph. For the remainder of this paper we assume that these are tracked at the level of individual instantiations, regardless of the instantiations of any enclosing generic types.

4.3 A Simple Example

Consider the following assemblies:

```
Assembly1.dll: Declares List<T>
Assembly2.dll: Declares class C
Assembly3.dll: Depends on 1,2, Uses List<C>
Assembly4.dll: Depends on 1,2, Uses List<C>
Assembly5.exe: Depends on 1,2,3,4, Uses List<C>
```

The pre-compilation strategy outlined above will generate one instantiation `List<ref>` in assembly 1. Furthermore if `C` is a value type, i.e. a type that is not representation-compatible with "ref", then further two instantiations `List<rep(C)>` are generated in assemblies 3 and 4. Assuming hard binding, no additional instantiations are placed in assembly 5 because all relevant instantiations can be found in the pre-compiled images for referenced assemblies.

4.4 Stable Homes for Reference Instantiations

As described above, the transitive-closure is computed with respect to the type representation function `rep`(T), and furthermore we speculatively pre-compile the (`<ref,...,ref>`) instantiation of each generic type `C<>` into the pre-compiled image of `C`. This gives a very useful guarantee: no additional code generation is ever required for instantiations of generic code at additional reference types. Thus reference instantiations are "cheap", and indeed this is a major argument in favour of the code sharing scheme described in [8].

The generation of (`<ref,...,ref>`) instantiations means that these are always compiled into a *stable home*, and will never be subject to the potential duplication effects described in Section 3.2.

4.5 Preferred Homes for Non-Reference Instantiations

The pre-compilation strategy can lead to the duplication of instantiations across multiple pre-compiled images. We now describe an important and novel optimization called *preferred home pre-compilation*.

In Section 3.1 we described how no single notion of a fixed home for a generic instantiation is possible in the presence of separate pre-compilation of assemblies. However, that need not stop us assigning a *preferred* home for each instantiation.

In particular, we assume we have a fixed, global function `PreferredHome` : *type* → *assembly* that assigns an assembly to each type. For example, `PreferredHome`(T) may depend only on the constituent atomic types involved in T itself. Many preferred home functions are possible, but for the rest of this paper we simply assume we take the assembly of the first atomic type that appears in a right-to-left scan of the structure of T. Note that for this function the preferred home of a type `List<C>` for an atomic type `C` will be the module of `C`, so preferred home functions can be chosen whose behaviour can be explained to programmers, at least in simple cases.

Our knowledge of the compilation status of each type descriptor w.r.t. its preferred home can then be classified as follows:

1. A type descriptor may have a *stable home*, i.e. the `<ref,...,ref>` instantiations described above.
2. Otherwise, type descriptors are assigned a *preferred home* according to a fixed, global scheme that depends only on the constituent atomic types involved in the type itself. At pre-compilation time a type descriptor may be known to be compiled into its preferred home. This may or may not be the current assembly being compiled.
3. Otherwise, we must assume we have no particular knowledge about a type descriptor, and must assume it will be either JIT loaded or compiled outside its preferred home.

References to type descriptors in categories 1 and 2 are eligible to be compiled as direct (perhaps relocatable) pointer references. References to other type descriptors are always compiled as symbolic link-time references.

We consider the performance implications of the preferred home optimization in Section 5.3.

4.6 The Linking Algorithm

At load-time, symbolic link-time references to type descriptors must be fixed-up as they are encountered. It is important to consider the link/loader logic that is required to ensure that this is sound when unique type descriptors are required within an implementation. This is achieved as follows:

1. If a type descriptor is a *stable home* descriptor then simply use the copy of the instantiation in that assembly's native image. If that assembly was not pre-compiled then JIT-load and compile the instantiation.
2. Otherwise, look in the preferred home of the instantiation. If the instantiation exists in that native image then use it.
3. Otherwise, consult a *dynamically linked descriptor* table to determine if a previous load operation has already resolved this load.
4. Otherwise, look through all existing native code images available. If the instantiation exists in any of these then use that instantiation and add that instantiation as an entry to the table from step 3.
5. Otherwise, JIT-load and compile the instantiation and add that instantiation as an entry to the table from step 3.

Together this approach establishes the following invariant: each compiled reference to a type descriptor, whether by a direct pointer or resolved by a load-time link operation, will ultimately be adjusted to refer to precisely the same unique item, whether that item was found in a native image or dynamically loaded. If an instantiation is pre-compiled into its preferred home then that copy of the instantiation will certainly be chosen as the canonical instantiation at runtime.

5 Performance Results

5.1 Case 1: A Compiler

Our first performance analysis relates to F♯, an implementation of an OCaml-like dialect of ML for .NET [16]. The benchmark is the bootstrapped F♯ compiler itself. In passing we note that command-line compilers represent a real challenge for JIT compilation as most code is executed even for small inputs. Pre-compilation seems particularly important for such applications.

The F♯ compiler compiles ML types to generic types such as `Func<A,B>` and `Pair<A,B>`. Furthermore the version of the F♯ compiler considered here performs no elimination of unnecessary ML polymorphism, even if a potentially polymorphic piece of code is used at only one instantiation, and thus ML's automatic generalization of functions creates a very large number of generic constructs. In all, the compiler acted as an extreme test-case for the performance of generic instantiations, containing 2239 declarations of type variables (on generic methods or classes), as opposed to 197 in the generic collections and related classes in the Microsoft .NET core library. The application exhibited a similar increase in the actual number of instantiations utilized at runtime.

Figure 5.1 shows the performance of the benchmark with and without pre-compilation, under a range of inputs. We note that the benchmark performs a significant amount of real computation at start-up, even for empty programs, as it reads preliminary data to build an initial type-checking context. It also builds in-memory representations of parser and lexer tables for the grammars processed by the compiler.

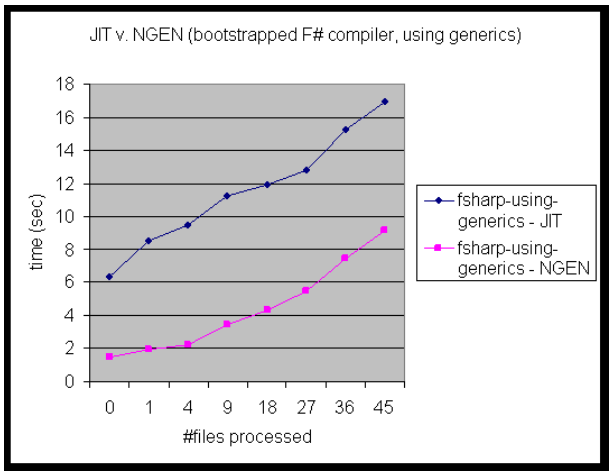

Fig. 1. Dramatically improved runtimes of a compiler benchmark over realistic inputs.

The figures indicate that "extreme" uses of generics can indeed lead to slow start-up times for applications of this kind. However, the data equally shows that pre-compilation of generic code can be used to amortize these costs. Furthermore, when the same compiler code base was compiled in other ways (e.g. by using an erasure model for generics, or by using the OCaml compiler) the resulting start-up time was roughly identical to that shown here.

5.2 Case 2: Collection classes

The C5 library is a publicly available implementation of a collection class library for C♯ featuring a variety of standard collection types implementing various interfaces representing different abstractions of collections. Based on an early version version of this library [15] we analyzed a client application called `FileIndex`, which indexes text files using `TreeMap` and `TreeSet` collection types.

The pre-compilation of the `FileIndex` application by the techniques described in Section 4 generates 66 ground instantiations of generic types. Of these, 35 are interface types that have no associated code. When the type-representation function `rep`(T) is applied to the remaining types 20 representative instantiations result, and of these 11 are reference instantiations. Thus there are 9 new representative instantiations at non-reference types.

Figure 5.2 shows the performance of a varying number of duplications of the FileIndex application under small inputs (we measure small inputs because we are most interested in start-up times). Here a "duplication" of the application refers to the duplication of the code of the application, but not of the collection classes themselves.

The diagram shows clearly the massive performance gain for NGEN in start-up (small input) scenarios. It also shows that the total committed memory usage (working set) suffers a minor relative degrade. This is related to the fact pre-compiled binaries contain more code and descriptors than are strictly required, as mentioned in the discussion on compilation granularity in Section 4.2. These additional constructs are paged-in when the truly required pre-compiled constructs are utilized.

5.3 Case 3: Page Sharing between Processes

Versions of the .NET Framework that support generics come with a namespace `System.Collections.Generic` that includes a type `List<T>` implementing simple expanding arrays. We performed an analysis of the marginal memory and image-size costs associated with pre-compiled instantiations of `List<T>` types in our prototype implementation.

The analysis involved a simple program that declared a varying number of instantiations of this type and performed some simple data-manipulation operations on single instances of each of these instantiations. The purpose of the experiment was to determine if the "preferred home" optimization from Section 5.3 was having the expected effect, i.e. that items compiled in the preferred module would incur low overhead, and in particular no writes to pre-compiled

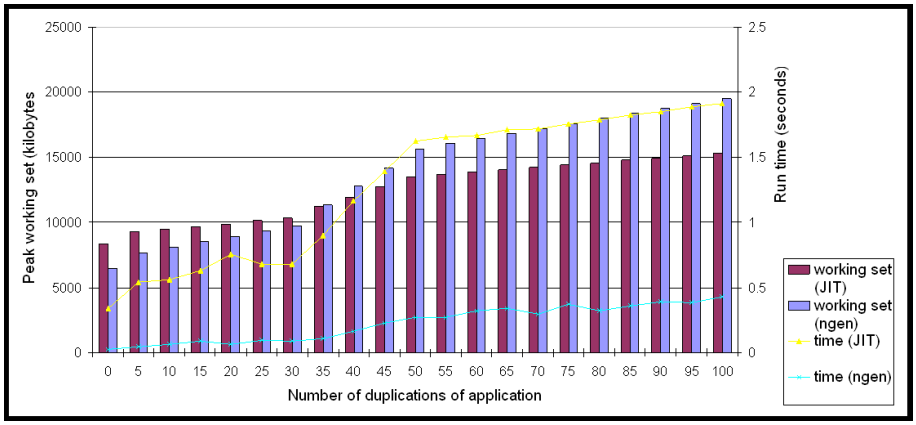

Fig. 2. A micro-benchmark simulating a series of generic-intensive components using generic classes defined in a library at identical instantiations. This represents a worst-case of the independent duplication of instantiations among different client components. Pre-compilation reduces start-up times to near-zero. Differential memory usage related to code is slightly higher in the pre-compiled case due to duplication, though 13 independent identical duplications are required before the initial memory reduction benefits of pre-compilation are lost.

images would be required to "fix up" these instantiations. We considered the following variables:

- Reference v. value-type instantiations;
- Instantiation chosen so that they would be pre-compiled into preferred v. non-preferred home modules;

We measured the average number of extra "touched" and "written" pages within the pre-compiled images for each new instantiation. Here a "written" page means that at least one write access was made to the given page of the pre-compiled binary image at runtime, significant because these pages may not be shared between multiple hardware processes and become "private" to each hardware process which makes such a write.

Kind	In Preferred Home?	Shareable Pages	Private Pages
`List<ref-type>`	Yes	0.12	0.0
`List<struct-type>`	Yes	0.27	0.0
`List<ref-type>`	No	0.04	0.15
`List<struct-type>`	No	0.41	0.21

The figures indicate that the "preferred home" optimization is indeed successful in eliminating all private pages to pages in pre-compiled images, and also in reducing the overal page usage. Items pre-compiled outside their preferred

home (or with the optimization disabled) incur reads and writes related to the linking performed during the loading process described in Section 4.6. Although these costs are one-off they are significant in start-up performance. Further ad-hoc experimentation confirmed that the marginal working set cost of additional instances of hardware processes executing the above program was indeed much lower with the optimization enabled. Finally, the code costs associated with non-reference instantiations are evident from the above figures, though that is to be expected given the nature of the compilation strategies taken for Generic IL.

6 Discussion and Related Work

In this paper we have described a set of techniques for pre-compiling .NET code that uses generics. The overall aim was to ensure that the start-up and memory savings achieved by .NET pre-compilation also apply to generic code, despite the inherent difficulties of combining separate compilation template-instantiation mechanism. The performance figures shown in Section 5 indicate that our goals have been achieved: reductions of up to 5x in start-up times have been observed, in line with other observed performance results for pre-compilation. Furthermore, aspects of the implementation have been transferred into the Microsoft .NET Framework 2.0 implementation of the CLR, resulting the first commercially significant programming languages to support both dynamic loading and type parameterization over unboxed/untagged representations.

The architectural choices involved in supporting a mixture of pre-compiled and dynamically-generated code are complex, and the design space is by no means fully explored. Other recent efforts include the "quasi-static compilation" supported by the QuickSilver system [14]. Other possible approaches include the more aggressive use of interpreted rather than native formats for generic code – for example, the OCaml bytecode compiler achieves good performance for a portable intermediate language without using any native compilation at all. Mixed native/interpreted systems have been explored in the context of implementations for many languages (e.g. see [5]). It is likely that native compilation is simply not optimal for all generic instantiations, and the widespread use of generics may require a reassessment of choices in this regard.

The techniques described appear to work best when we can assume a number of things about the generic code being executed. Firstly, an assumption is made that code is the expensive thing to create at runtime, i.e. that exact-descriptors for types can be generated and stored efficiently at runtime: we have not yet seen this assumption contradicted in practice. Secondly, the techniques will generate strictly fewer copies of instantiations when two assemblies are merged, and thus the technique performs better when the unit of pre-compilation is fairly large, i.e. if compilation is performed at the granularity of assemblies rather than at the granularity of individual classes.

The problems of supporting the combination of Generic IL and pre-compilation are reminiscent of the template-management problem for C++ compilers and linkers: where are template instantiations stored, when are they generated, and

how are intermediary compilands stored? On-disk caches have been used for the compilation of C++ templates [11], the main aim being to reduce compilation times associated with repeated template instantiations.

The overall problem of code and resource sharing in JVMs has been addressed from a number of perspectives, and an excellent comparative study can be found in [6]. The techniques described there are very different to those used for .NET pre-compilation, and the erasure model of Java generics means that no interaction between generics and pre-compilation arises. Additional aspects of this interaction relating to application domains have been explored in [9]. Related issues arise in attempts to use virtual machines as the basis for operating systems [3, 2].

6.1 Acknowledgements

We would like to thank Sean Trowbridge, Dario Russi, Jim Hogg, Patrick Dussud and Claudio Russo for helpful discussions related to this work.

References

1. Sun Microsystems Feedback Page. See `http://bugs.sun.com/`, Bug number 4607280. As of Oct 2005 comments included *Perl takes 0.03sec to launch and print. Python takes 0.1sec. Java 1.4 takes 3.0 sec.* and *slow loading is probably one of the biggest reasons to avoid the use of Java applets*.
2. 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.
3. 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.
4. 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.
5. M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.
6. G. Czajkowski, L. Daynès, and N. Nystrom. Code sharing among virtual machines. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 155–177, London, UK, 2002. Springer-Verlag.
7. ECMA International. Ecma standard 334: C# language specification. Available at `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.
8. 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.
9. A. J. Kennedy and D. Syme. Combining Generics, Pre-compilation and Sharing Between Software-Based Processes. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE), January 2004.

10. A. J. Kennedy and D. Syme. Transposing F to C#: Expressivity of parametric polymorphism in an object-oriented language. In *Concurrency and Computation: Practice and Experience*, to appear.
11. J. R. Levine. *Linkers and Loaders*. Morgan-Kauffman, October 1999.
12. Microsoft Corporation. An Introduction to C# Generics. See website at `http://msdn.microsoft.com/vcsharp`.
13. 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.
14. 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.
15. P. Sestoft. Generic C# sample programs, 2001. See `http://www.dina.kvl.dk/~sestoft/gcsharp/`.
16. D. Syme. The F# programming language. See `http://research.microsoft.com/projects/fsharp`.