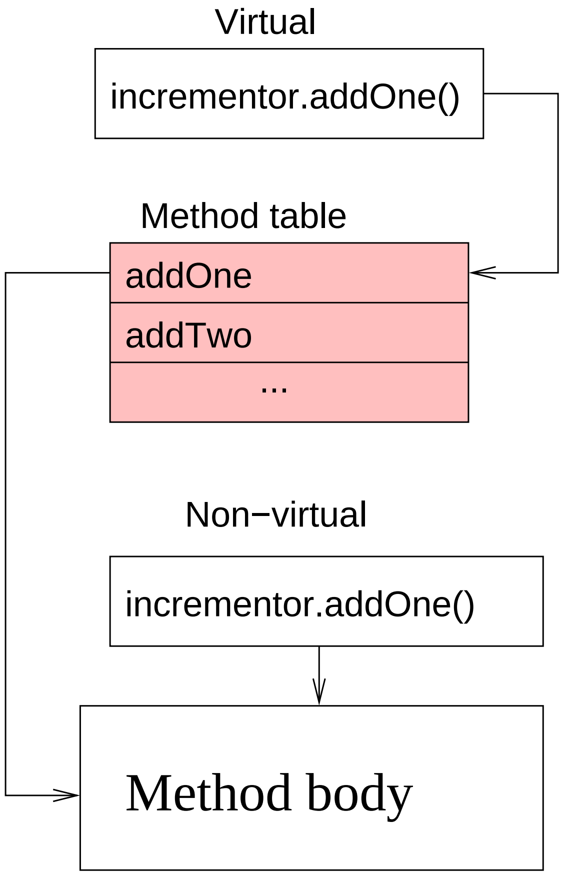
A Method Specialisation and Virtualised Execution Environment for Java

Andrew M Cheadle
Anthony J Field
Johan T Nyström-Persson

Imperial College London

Dynamic Dispatch




```
public class Incrementor {
  int value = 0;
  public void addOne() {
    value++;
  }

  ...
}
```

Partial: Java

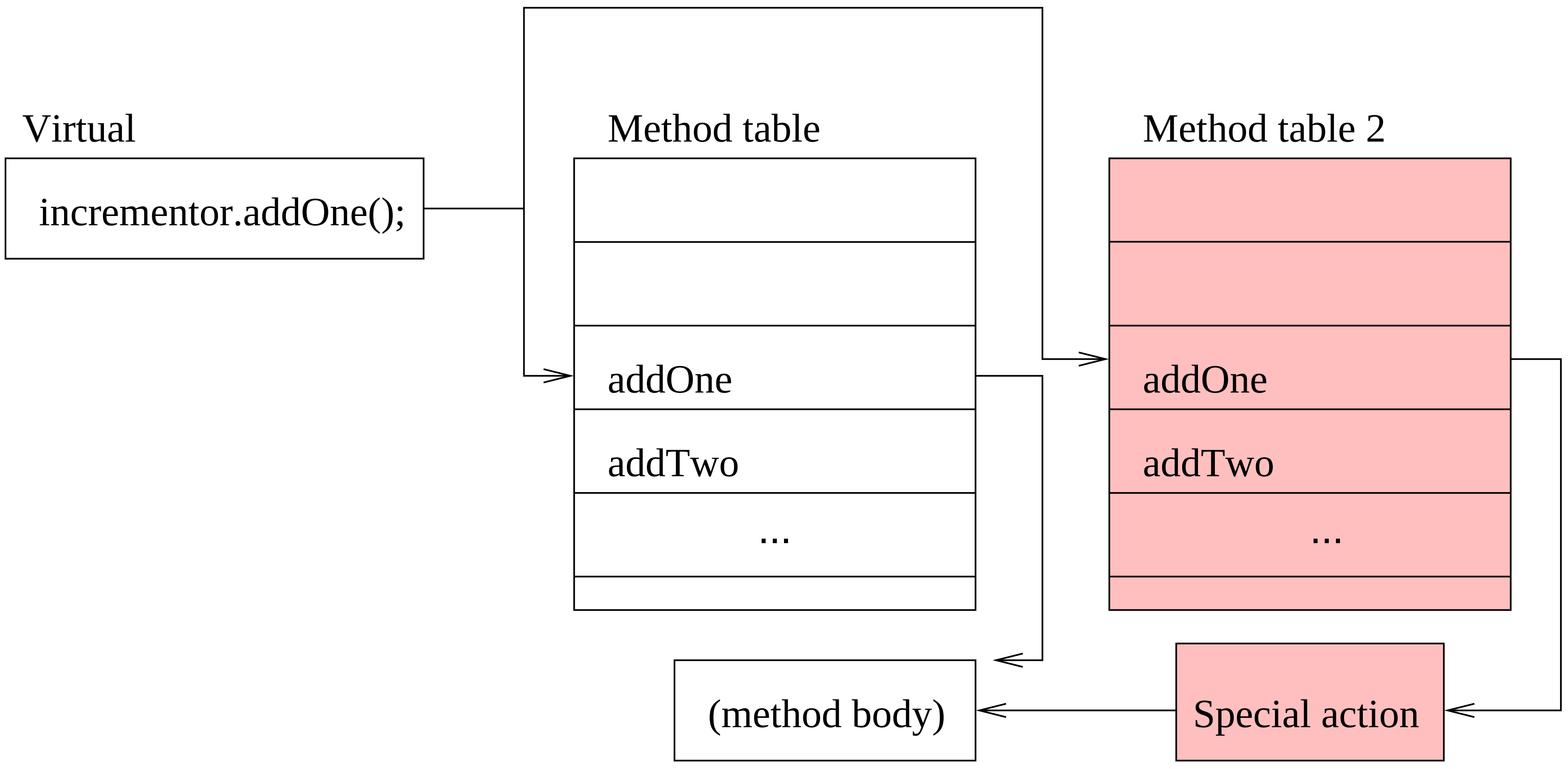
Optional: C++, C#

Complete: Message dispatch in Smalltalk

Idea

- Virtualise *everything*
- Specialise anything
- Opposite of conventional wisdom for optimizing dynamic languages
- Buy back resulting overhead by exploiting the indirection

Exploiting Virtualisation



Applications

- Object Proxying
- Orthogonal persistence
- Distributed JVM: ANU's dJVM, IBM's cJVM
- Profiling
- Instrumentation
- Fickle objects
- *Read barrier for incremental garbage collector*

This Paper

- Framework for *full* virtualisation and method specialisation.
- Optimization is allowed to proceed using guarded inlining of specialisations.
- Case Study - Elimination of redundant incremental garbage collector read barriers
- Evaluation of both framework and dynamic dispatching read barrier overheads

Class Transform Toolkit (CTTk)

- Our framework for virtualisation and specialisation
- Uses *Jakarta BCEL* to modify and generate Java code
- Pipelined code transformations

The CTTk framework API

Class Transformation Toolkit

```
public interface Transform {
  public TransformClass transform(TransformClass transformClass);
}

public class TransformClass {
  public void setJavaClass(JavaClass javaClass) { ... }
  public JavaClass getJavaClass() { ... }
  public void setSpecialisedMethods(Method[][] specialisedMethods)
   { ... }
  public Method[][] getSpecialisedMethods() { ... }
}
```

VMT switching API

```
VM_ObjectModel.restoreTIB(Object obj);

VM_ObjectModel.hijackTIB(Object obj, int tibSpec);
```

Virtualisation

- Effectively *beanification* - Java Bean contract
- Achieve full virtualisation through code transformation
- Create access methods for each field
- Replace each direct field access with a call to an access method
- Remove redundant access methods

Virtualisation - effects

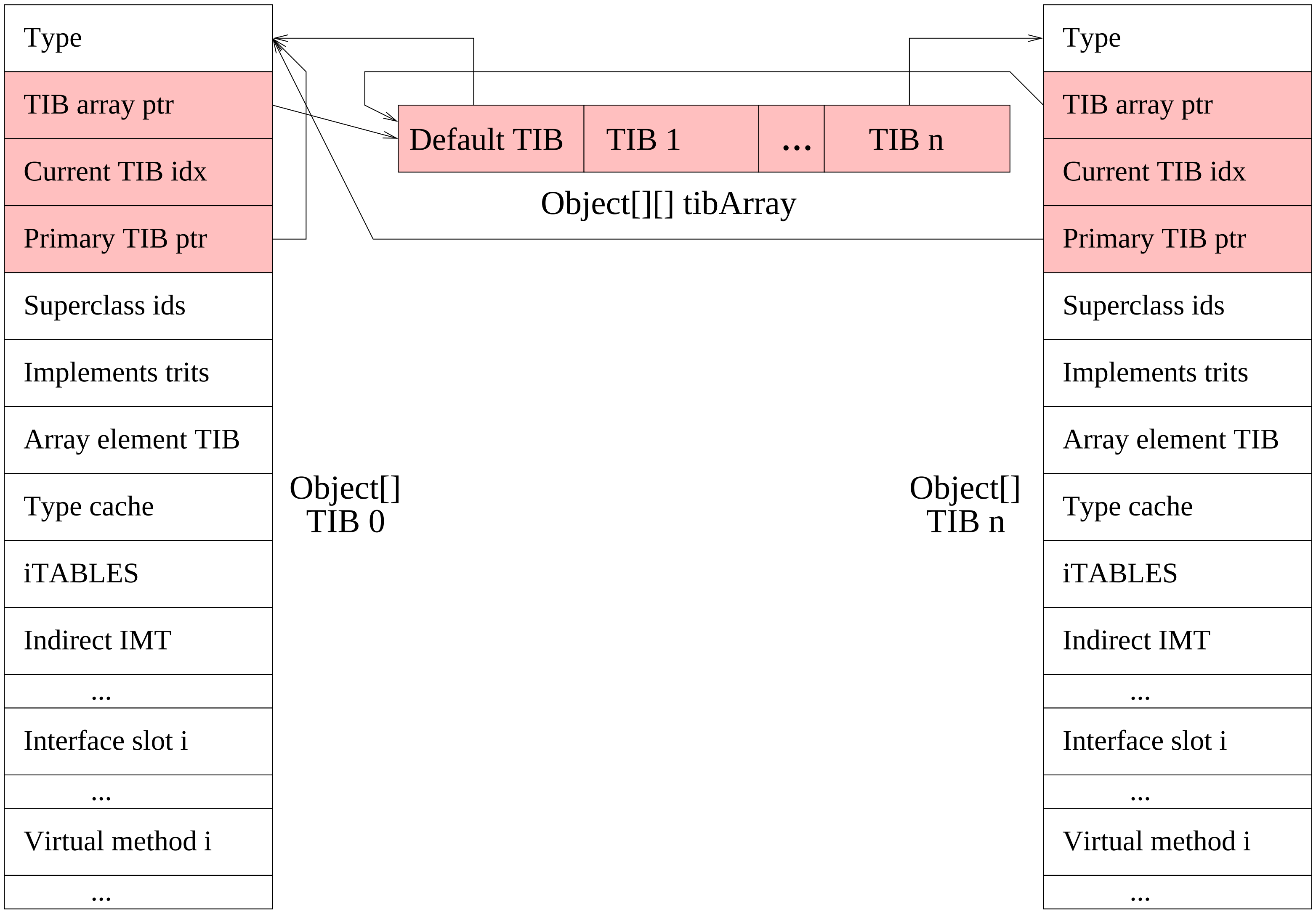
```
public class Incrementor {
  private int value = 0;
  int _g_value_Incrementor_I() {
    return value;
  }
  void _s_value_Incrementor_I(int field1) {
    value = field1;
  }
  public void addOne() {
      _s_value_Incrementor_I(_g_value_Incrementor_I() + 1);
  }
}
```

The Type Information Block

Type
Superclass ids
Implements trits
Array element TIB
Type cache
iTABLES
Indirect IMT
...
Interface slot i
...
Virtual method i
...

- The TIB is an Object array
- Full type description
- Contains method table!

TIB specialisations



Inlining

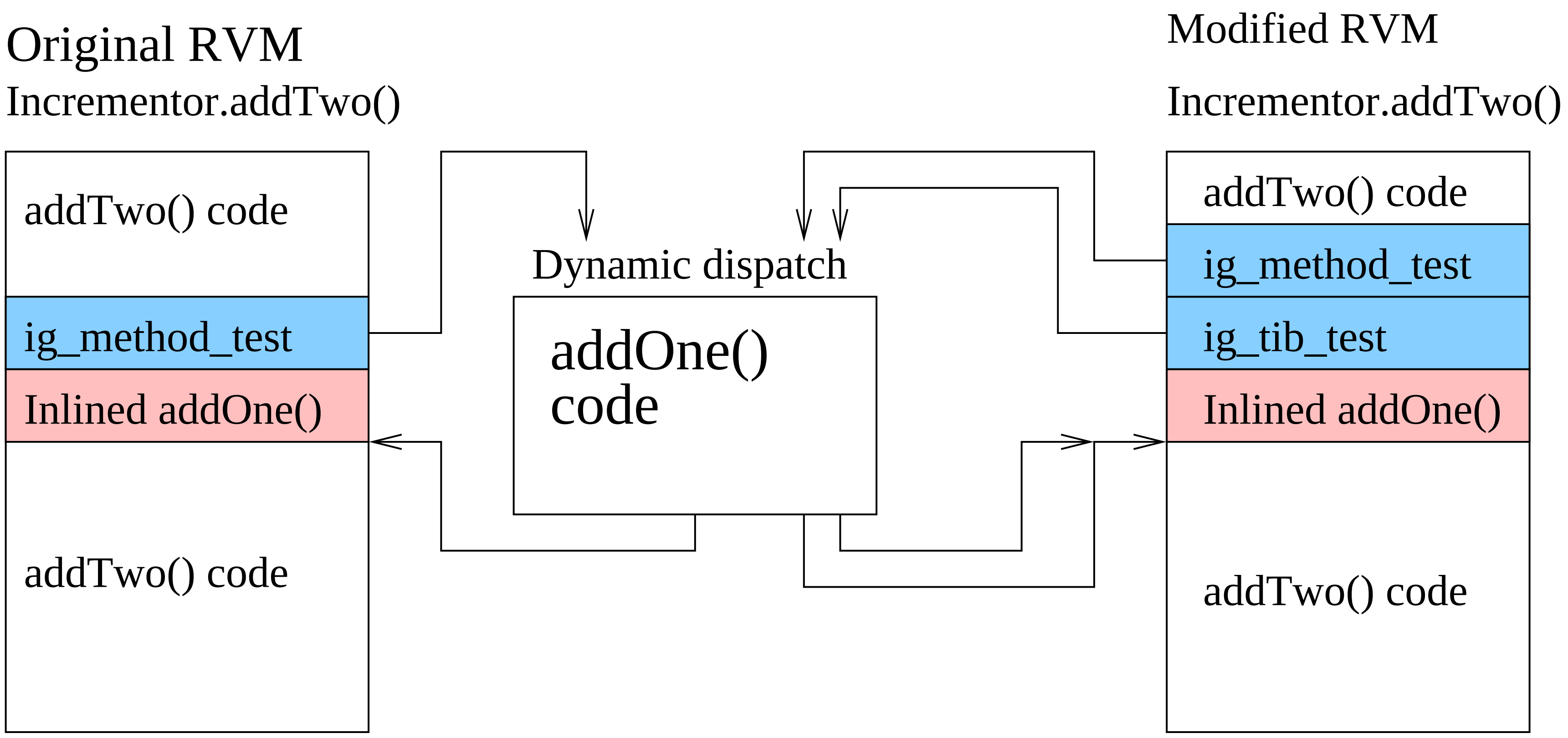
```
public class Incrementor {
  int value = 0;
  public void addOne() {
    value++;
  }
  public void addTwo() {
    addOne();
    addOne();
  }
}
```

```
public class Incrementor {
  int value = 0;
  public void addOne() {
    value++;
  }
  public void addTwo() {
    //inlined call site
    value++;
    //inlined call site
    value++;
  }
}
```

Inline guards

- Inlining is a crucial optimization
- Call site is provably *monomorphic*: inline directly
- Call site is *polymorphic*: perform *guarded inlining*
- Typically one of: `ig_method_test`, `ig_class_test`, `ig_patch_point`
- We add the `ig_tib_test` and allow for an arbitrary number of guards (we use at most 2)

Inline Guards in Practice



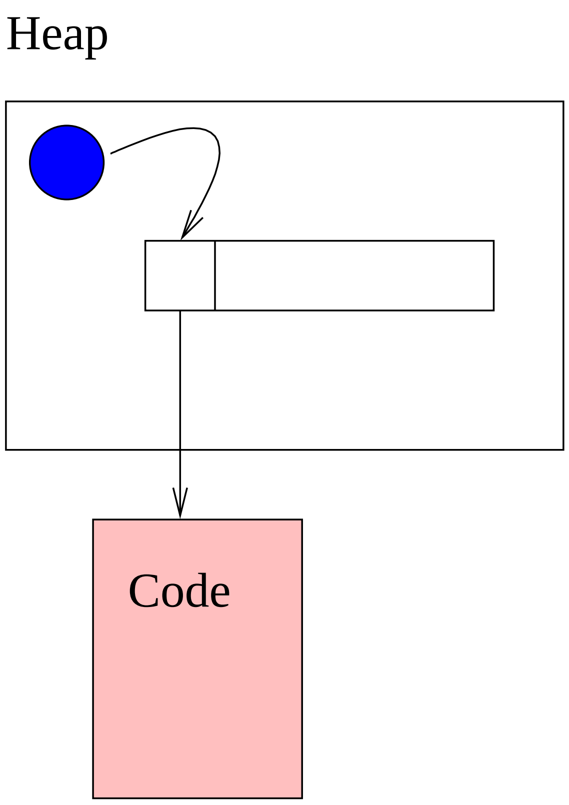
If any of the guard tests fail, the method will be invoked as normal.

Case Study: Read Barrier for GC

- Use CTTk to create the *IncrGCTransform*
- Transform creates specialised methods: 1 normal TIB, one specialised
- Specialised methods maintain the to-space invariant; they scavenge the object, then restore the TIB and invoke the method as normal
- The read barrier action is thus inserted at no extra cost, *only where needed*

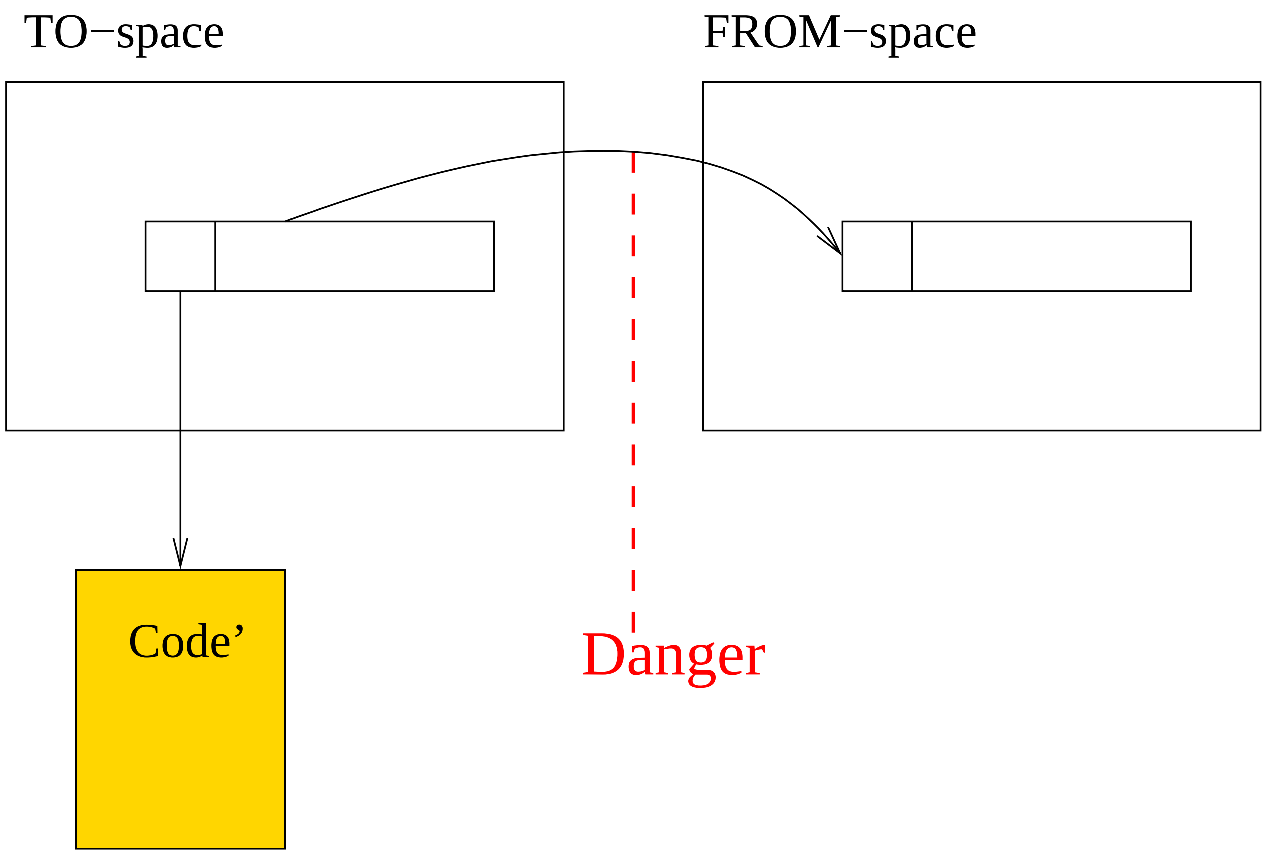
Read Barrier for GC - Summary

a. GC off



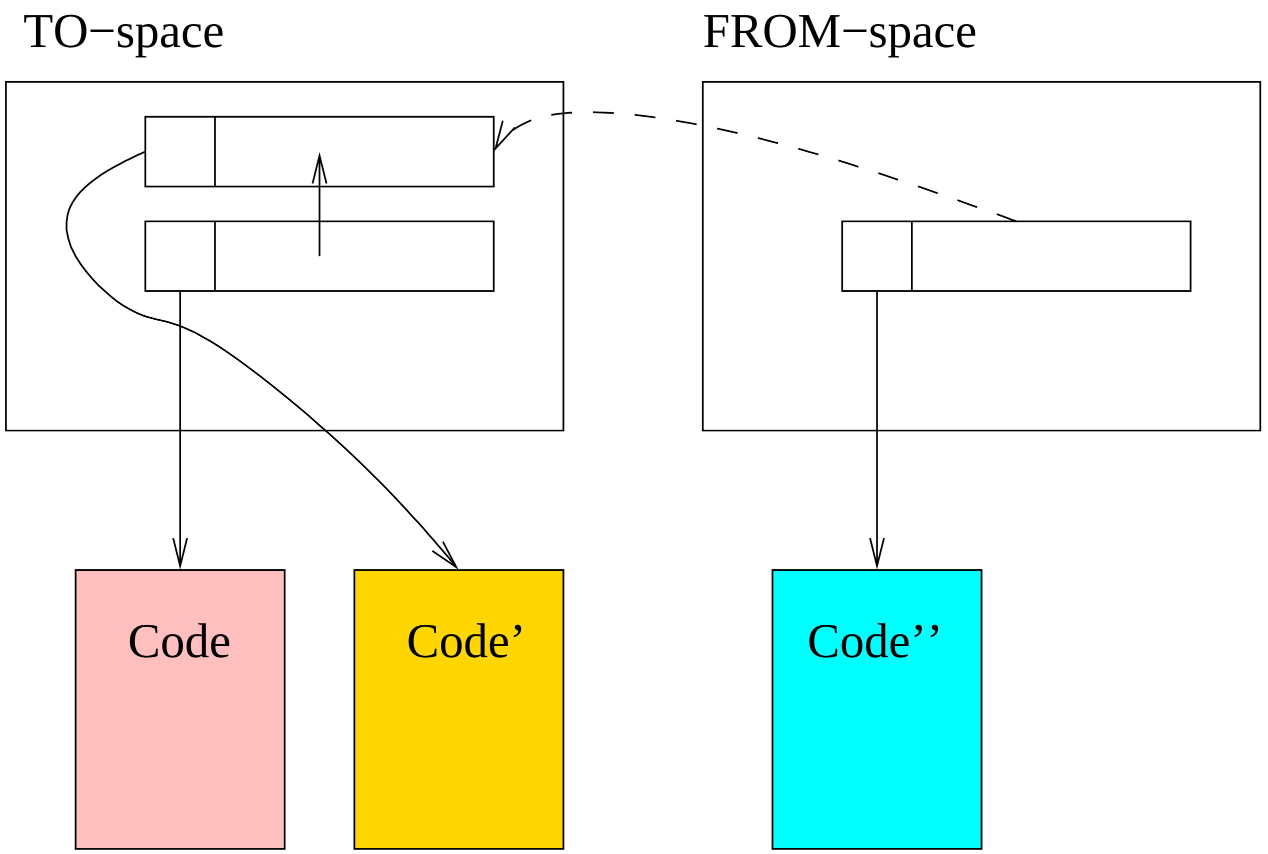
Read Barrier for GC - Summary

b. GC on - copy all objects into TO-space



Read Barrier for GC - Summary

c. Reference accesses object



Generated specialisations

```
public class Incrementor {
  int value = 0;
  public void addOne[0]() {
    _s_value_Incrementor_I(_g_value_Incrementor_I() + 1);
  }
  public void addOne[1]() {
    VM_ObjectModel.restoreTIB(this);
    MMTk.scavenge(this);
    addOne(); // will invoke addOne[0]
  }

}
```

Concurrency

- Our framework ensures that no thread switchpoints are being emitted while we switch between one TIB and another.
- However, we make no guarantee about the order in which methods or specialisations of methods will be executed. make sure that their code is thread safe.
- Users must define and enforce a policy by themselves.

Evaluation

- Evaluate read barrier overheads using upper and lower bounds
- Upper bound — all objects scavenged via the mutator using method specialisations
- Lower bound — the collector scavenges all objects, the mutator none
- DaCapo and Spec JVM98 benchmarks run to convergence

CTTk Overheads

CTTk overheads on execution time

Benchmark	Original(s) Init	Original(s) Conv	Virtual (% o/head) Init	Virtual (% o/head) Conv
_201_compress	6.17	5.02	42.63%	8.37%
_202_jess	3.54	2.83	90.40%	2.83%
_209_db	13.72	13.40	11.88%	2.69%
_213_javac	8.83	5.87	91.50%	21.64%
_222_mpegaudio	6.68	3.72	28.44%	5.65%
_227_mtrt	5.08	2.53	30.31%	7.11%
antlr	5.57	4.46	328.17%	5.38%
bloat	15.02	11.51	165.79%	11.99%
fop	5.33	3.21	711.82%	3.74%
luindex	19.36	15.40	273.46%	21.49%
pmd	15.14	9.78	360.69%	20.96%
Min	3.54	2.83	11.88%	2.69%
Max	19.36	13.4	711.82%	21.49%
Geo. mean	8.23	5.82	102.97%	7.75%

Read Barrier Overheads

Benchmark	Implicit (Lower) Init	Implicit (Lower) Conv	Implicit (Upper) Init	Implicit (Upper) Conv	Explicit Init	Explicit Conv
_201_compress	42.79%	8.37%	42.86%	8.37%	24.47%	43.63%
_202_jess	90.89%	2.95%	91.11%	3.00%	7.62%	7.42%
_209_db	12.10%	2.86%	1.12%	2.93%	12.39%	12.31%
_213_javac	59.52%	23.27%	64.35%	23.99%	5.55%	4.94%
_222_mpegaudio	28.62%	5.66%	28.70%	5.66%	48.20%	34.95%
_227_mtrt	42.79%	8.21%	31.34%	8.70%	2.17%	32.02%
antlr	328.17%	5.62%	328.17%	5.72%	11.67%	8.30%
bloat*	165.79%	12.27%	165.79%	12.41%	21.17%	18.11%
fop*	711.83%	4.41%	711.83%	4.41%	32.80%	28.40%
luindex*	273.46%	21.49%	273.46%	21.49%	22.74%	19.03%
pmd*	360.70%	21.0%	360.70%	21.04%	8.11%	7.22%
Min	12.10%	2.86%	1.12%	2.93%	2.17%	4.94%
Max	711.83%	23.27%	711.83%	23.99%	48.20%	43.63%
Geo. mean	102.49%	8.15%	80.87%	8.27%	13.13%	15.56%

* For the explicit barrier, not all benchmarks can be run to completion. Reported values are based on measurements taken prior to the crashes.

(De)Virtualisation Count/Bloat

(De)Virtualisation Counts and (Static) Code Bloat

Benchmark	Virt^{n}s	Devirt^{n}s	Code Bloat(%)
_201_compress	5189	2989	10.1
_202_jess	5977	3684	13.7
_209_db	5078	2564	23.2
_213_javac	9159	2668	11.8
_222_mpegaudio	6172	3673	32.2
_227_mtrt	5499	2635	28.1
antlr	14025	4716	21.2
bloat	9871	9781	12.1
fop	9295	3431	24.2
luindex	6661	2513	18.2
pmd	13898	2928	26.3

BCEL Overheads

Benchmark	BCEL only (Conv)
_201_compress	7.57%
_202_jess	2.47%
_209_db	1.27%
_213_javac	13.12%
_222_mpegaudio	2.42%
_227_mtrt	3.95%
antlr	4.26%
bloat	8.95%
fop	3.12%
luindex	14.09%
pmd	14.83%
Min	1.27%
Max	14.83%
Geo. mean	5.19%

Conclusions

- Overhead of full virtualisation is low
- CTTk delivers both ease of use and good performance
- A read barrier for an incremental collector has been constructed with relative ease, and it compares favourably

Work in progress

- Framework is now current with Jikes RVM CVS Head and Java 1.5 -> Spec JBB.
- Currently implementing incremental collectors for Jikes RVM
  - Explicit Baker-style read barrier collector
  - Dynamic dispatching read barrier collector
- Porting the framework to use ASM (more lightweight) instead of BCEL
- DSL (syntactic sugar) to ease transform specification