Supporting Java for soft real-time interactive systems

Tim Harris

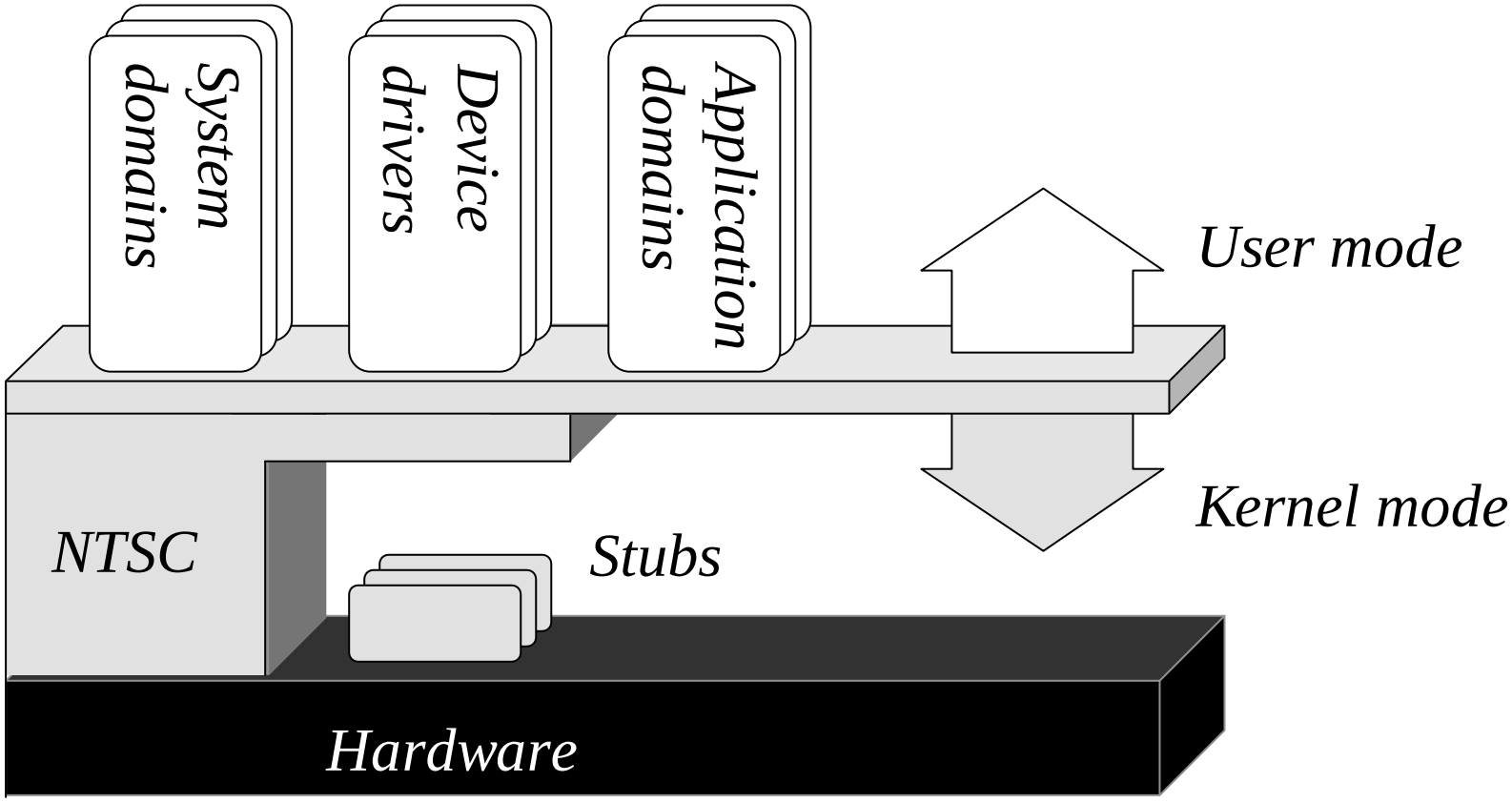
Introduction

- Implementing the JVM well is a hard problem:
  - Interpreting code gives poor performance
  - Compiling code can introduce uncontrollable pauses
  - Cross-talk between different threads in the JVM

Nemesis

- An operating system designed to support soft real-time multimedia applications
- Provides fine-grained control over resources
  - eg CPU, disk bandwidth, network bandwidth
- Provides resource-accountability by avoiding shared servers and processing within the kernel
  - Network protocols are implemented in user-level libraries
  - “Client renders” window system
- Security can still be enforced

Structure of Nemesis



Java thread scheduling

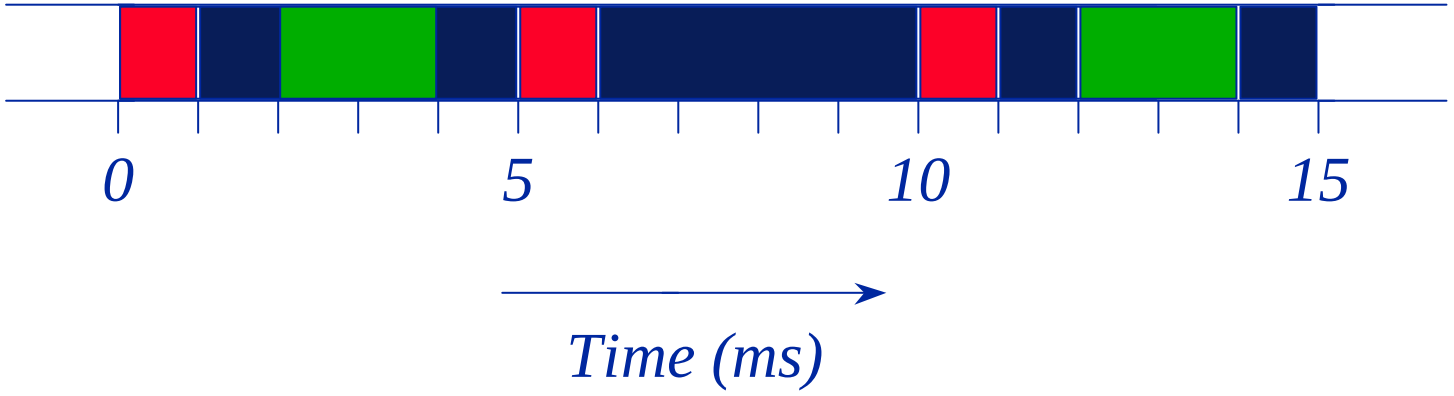
- Java normally uses priority hints to control thread scheduling policy
  - Inflexible way of expressing CPU requirements
  - Choosing a priority is hard
  - Low priority threads can be permanently starved
  - Priority inversion is a problem

New thread scheduler

- New scheduler expresses CPU requirements as:
  - Period & slice
  - Extra time flag
- Can accommodate a variety of tasks, eg:
  - 10% every 1ms — interactive program
  - 5% every 100ms — background task
  - extra time only — unimportant task
- Extra time is shared out according to priority hints

New thread scheduler (2)

- Thread A: 1ms every 5ms
- Thread B: 2ms every 10ms
- Thread C: 1ms every 5ms, and any extra time



Run-time compilation - problems

- Run-time compilation is necessary for acceptable performance
  - Many optimizations in Java can only be made at run time
- Simple ‘Just in time’ compilation introduces uncontrollable pauses
  - Initialization methods are only executed once and are often large
  - Forces compiler to operate quickly and to omit important optimizations
  - Compiled code is thrown away when the program exits

Run-time compilation

- Provide mechanisms which allow the programmer to control the compiler and to implement their own policy, eg:
  - Compile quickly on first invocation
  - Compile with maximum optimization
  - Compile in the background
  - Never compile
- Provide a default policy for other applications

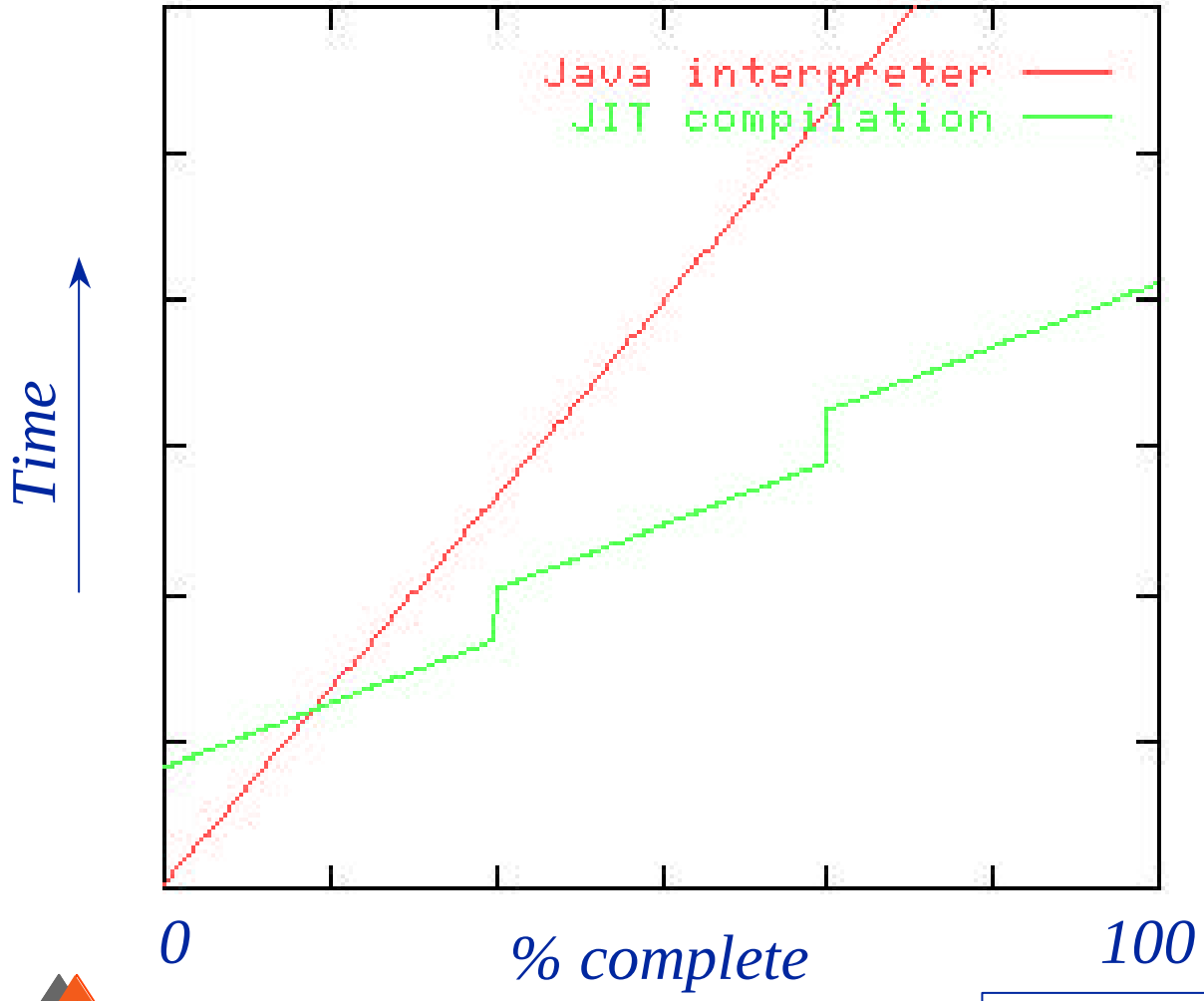
Compilation dispatchers

- A *dispatcher* class is associated with a region of the package hierarchy, eg:
  - `*` ⇒ Compile in the background
  - `java.lang.*` ⇒ Load pre-compiled native code
  - `myapplet.*` ⇒ Compile on first invocation
  - `myapplet.UserInterface` ⇒ Never compile
- The most specific match is chosen

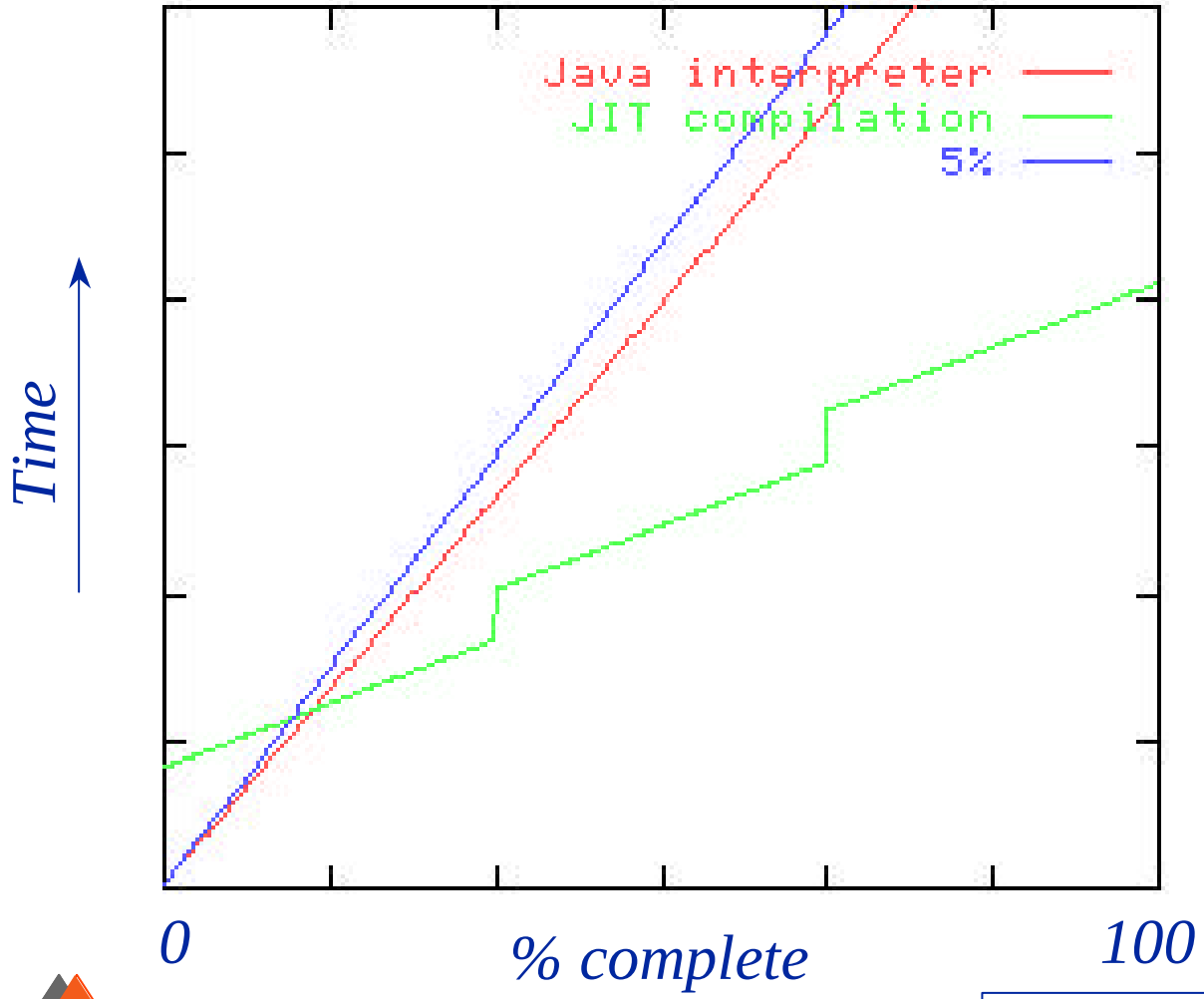
Background compilation

- One thread compiles a method, while another starts interpreting it
- Control CPU allocation to the compilation thread:

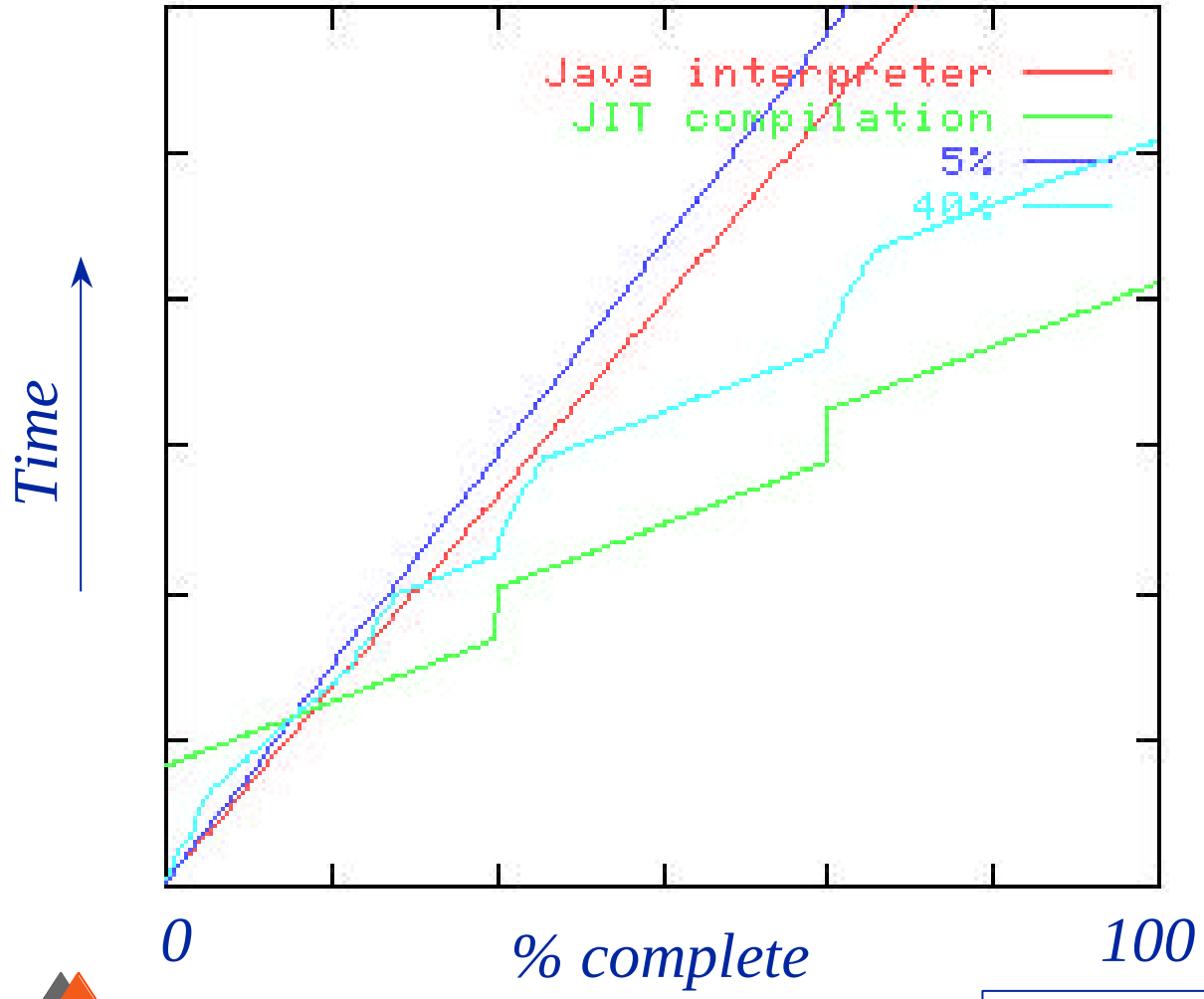
Background compilation



Background compilation



Background compilation

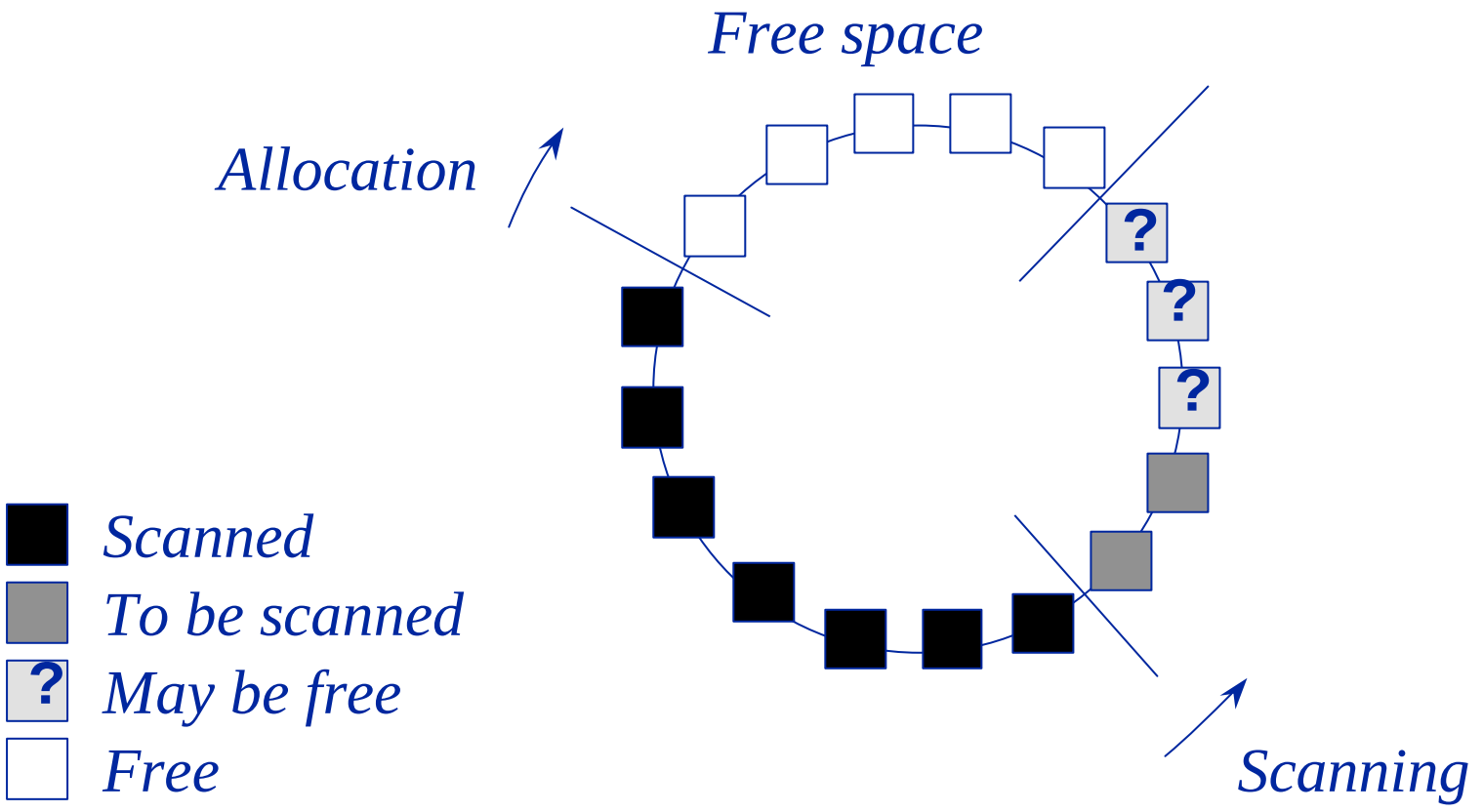


Garbage collection

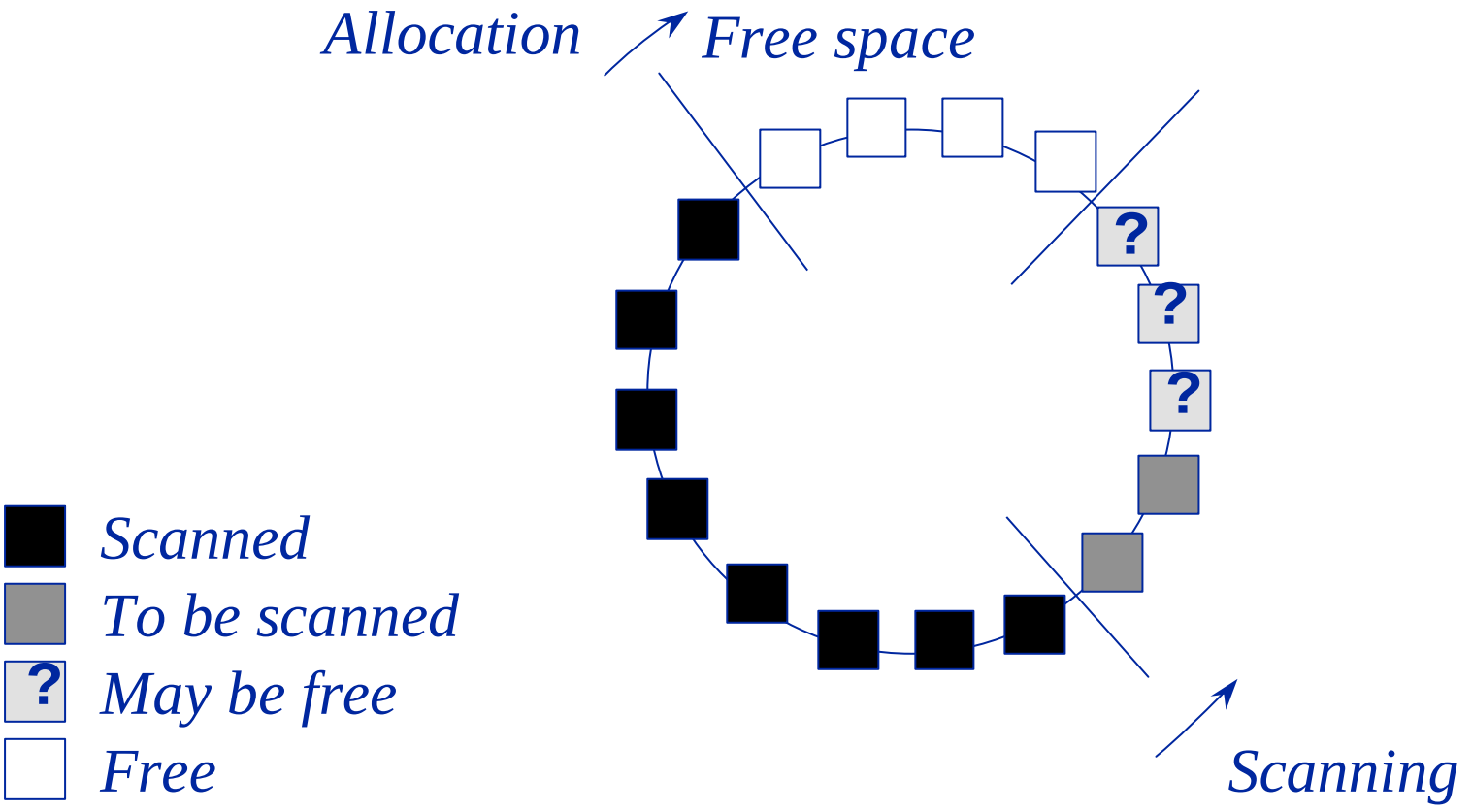
- Initial implementations have used non-concurrent collectors which cannot be interrupted
- A source of cross-talk, eg:

```
while (true)
{
     System.gc();
}
```

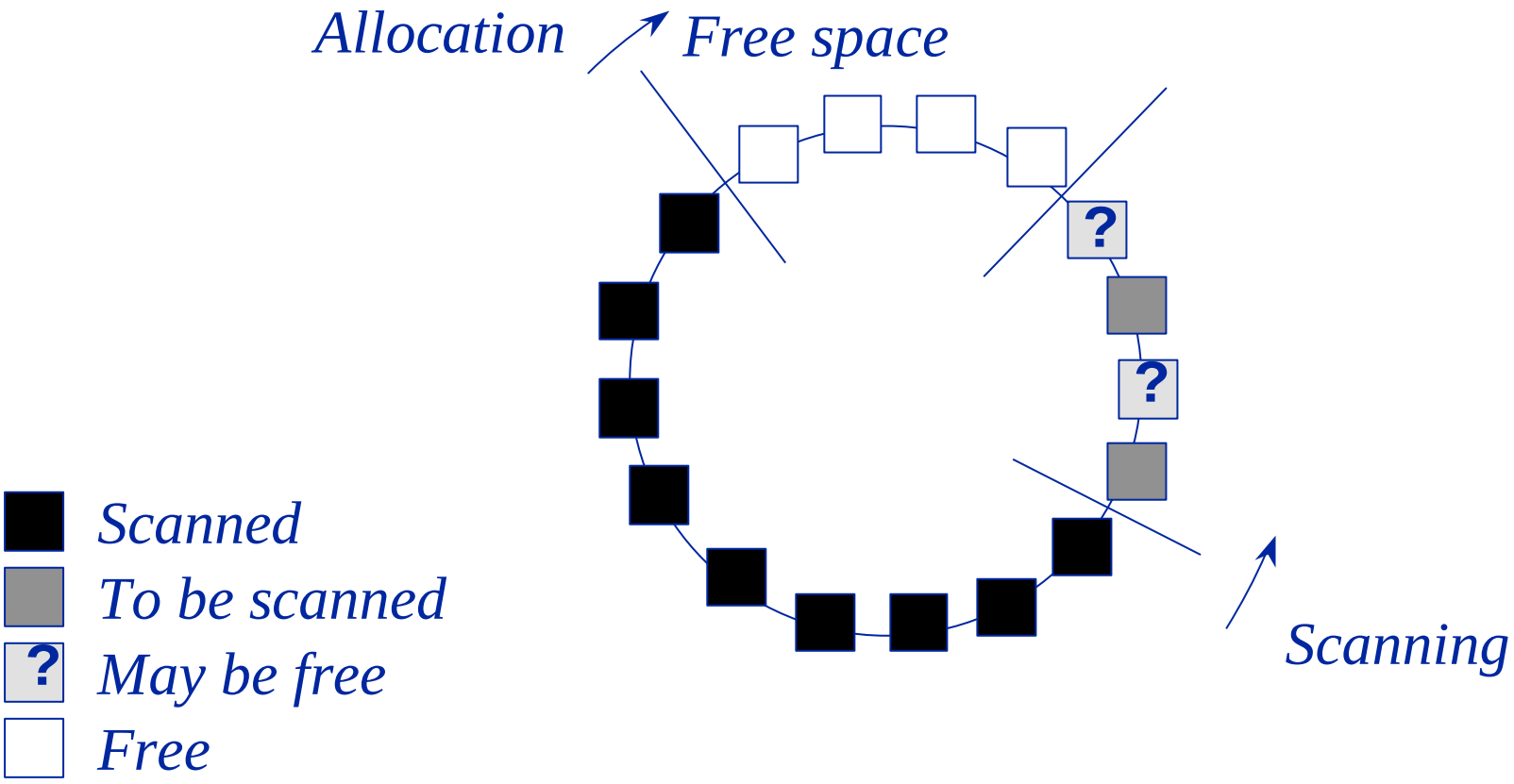
Baker's treadmill collector



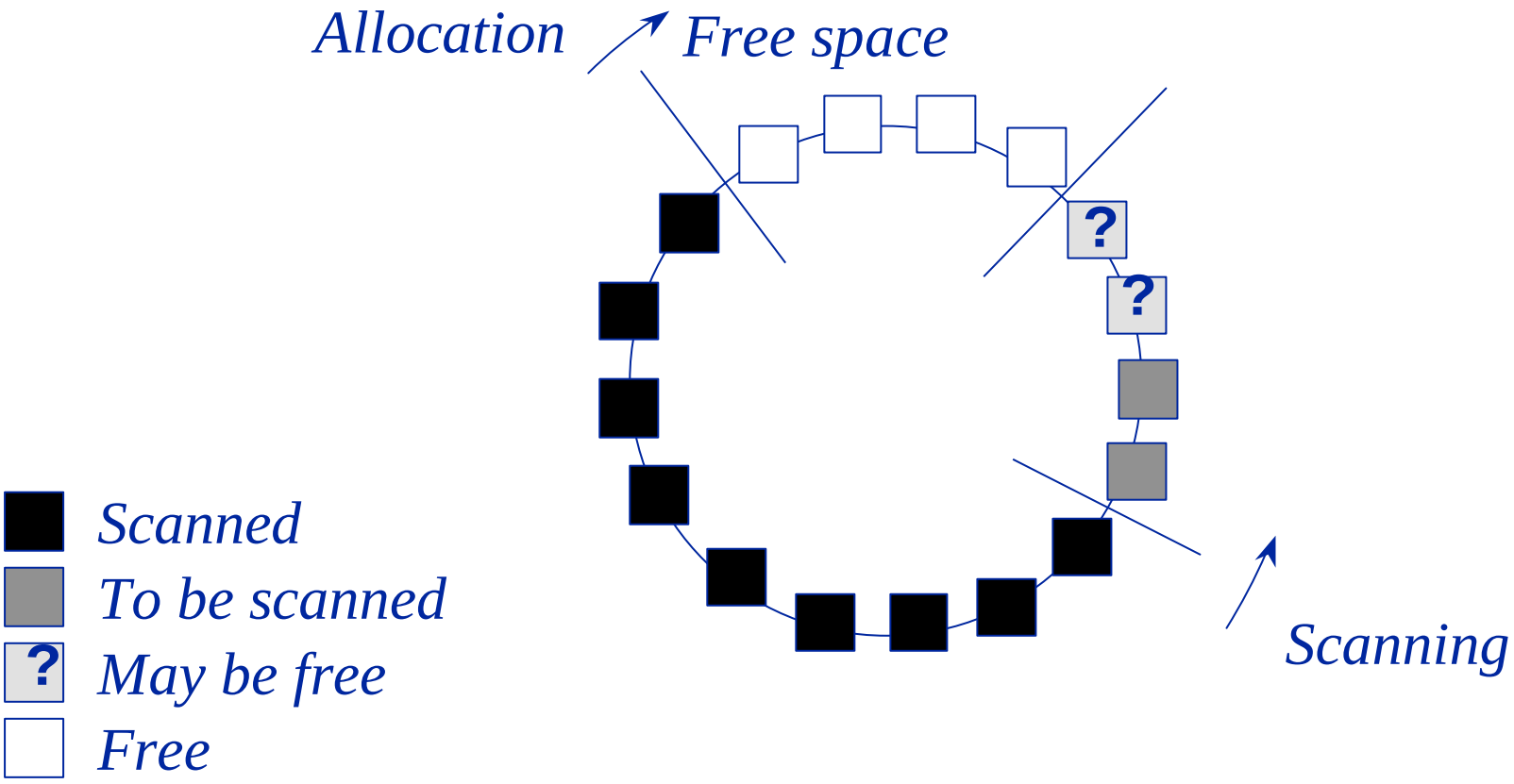
Baker's treadmill collector



Baker's treadmill collector



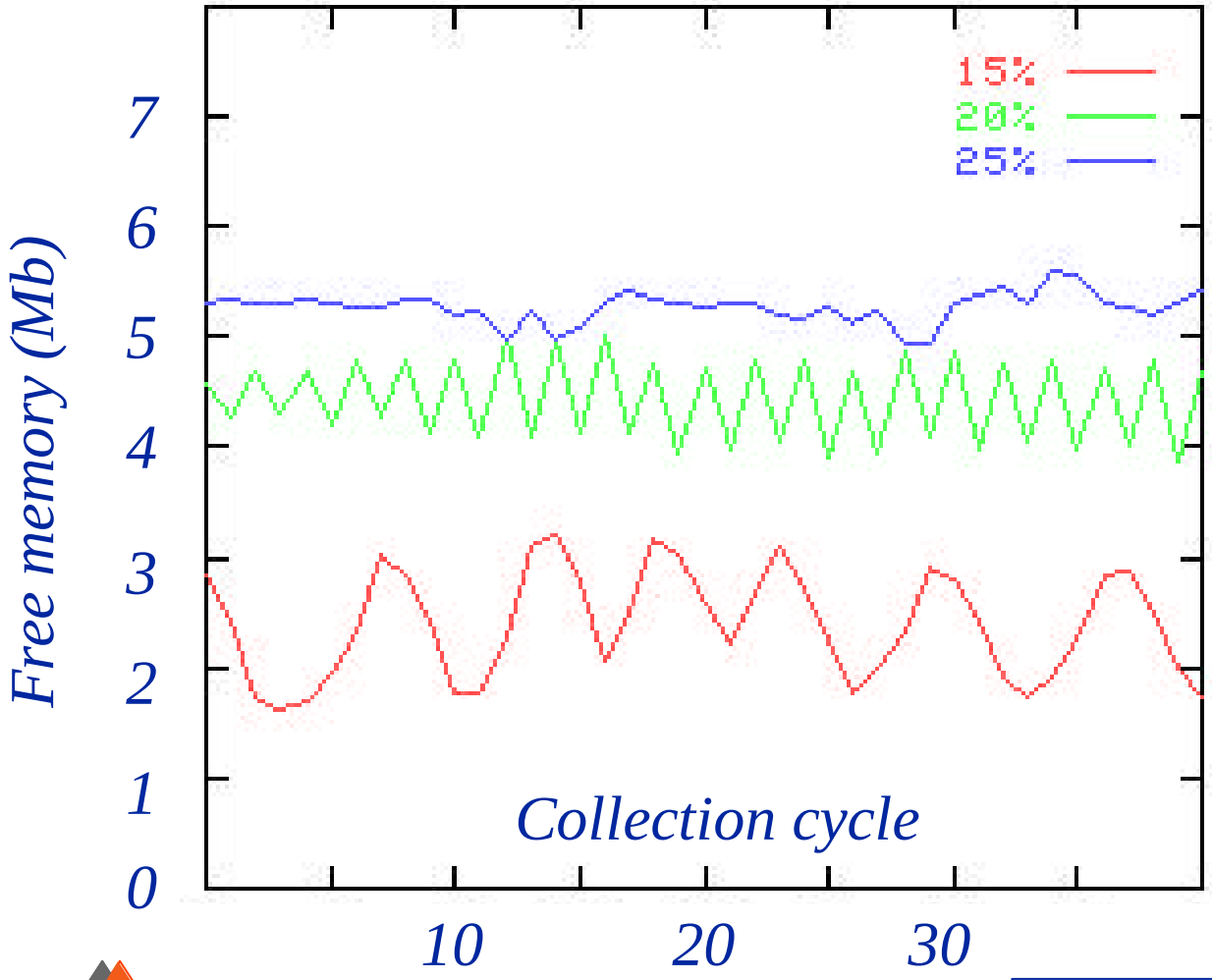
Baker's treadmill collector



Garbage collection

- Allocation and collection can continue in parallel
- Sufficient CPU time must be allocated to the collector so that it finishes scanning before free space is exhausted
- Can trade time spent collecting against:
  - Probability of blocking an allocation
  - Memory required

Garbage collection



Conclusions & future work

- Java requires more runtime support than other popular languages
- JVM implementations are large and enforce many inflexible policies
- JVM is closely tied to the Java language definition

Take a “Nemesis-like” approach by providing only the essential mechanisms for security and sharing resources