



Extensible virtual machines



3 May 2001

tim.harris@cl.cam.ac.uk

Extensible virtual machines



- The aim here is to provide *untrusted* programmers with fine-grained control over how their code is executed
- ... it's not specifically about
 - supporting multiple languages
 - direct hardware access
 - making things go fast

Conventional VMs



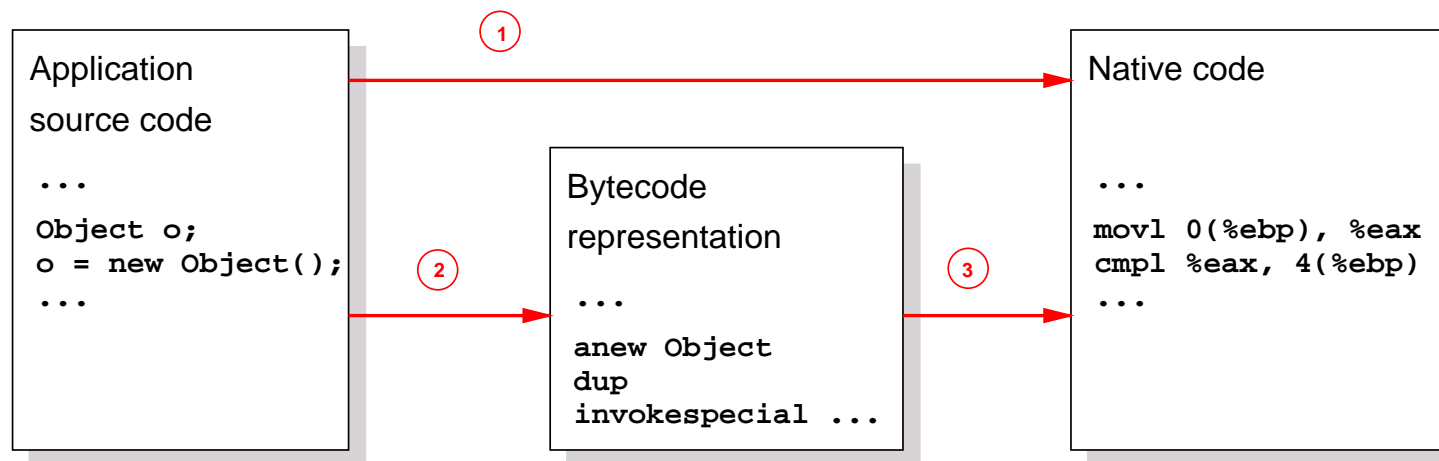
- VMs for ‘safe’ languages typically provide abstraction from machine-specifics and much much more:
 - run-time compilation from bytecode
 - storage allocation and garbage collection
 - selection of object representations
 - extensive libraries
 - ...
- Unlike with (e.g.) C the programmer can’t change these

Motivation



- Judicious decisions about (e.g.) storage management, compilation, scheduling depend critically on application behaviour
 - copying GC performs well if few objects survive: $O(\textit{retained})$ rather than $O(\textit{freed})$,
 - when should compilation occur – would like it to unobtrusive and any run-time stats it needs to be gathered cheaply,
 - frequent thread context switches may aid responsiveness... but each switch has a cost.

Motivation (2)



- Existing (safe) bytecode formats simply can't implement e.g. object allocation in a useful way
- What can we do instead?

Overview



- Separate the implementation of the
 - *protected mechanisms* which are currently most readily implemented in trusted native codefrom that of the
 - *untrusted policies* which can be provided by the application programmer
- also, encourage separation between policies and the applications themselves: allow policy/application re-use. The mapping is maintained by the *policy registry*.

Overview (2)



Problem domain: storage allocation,

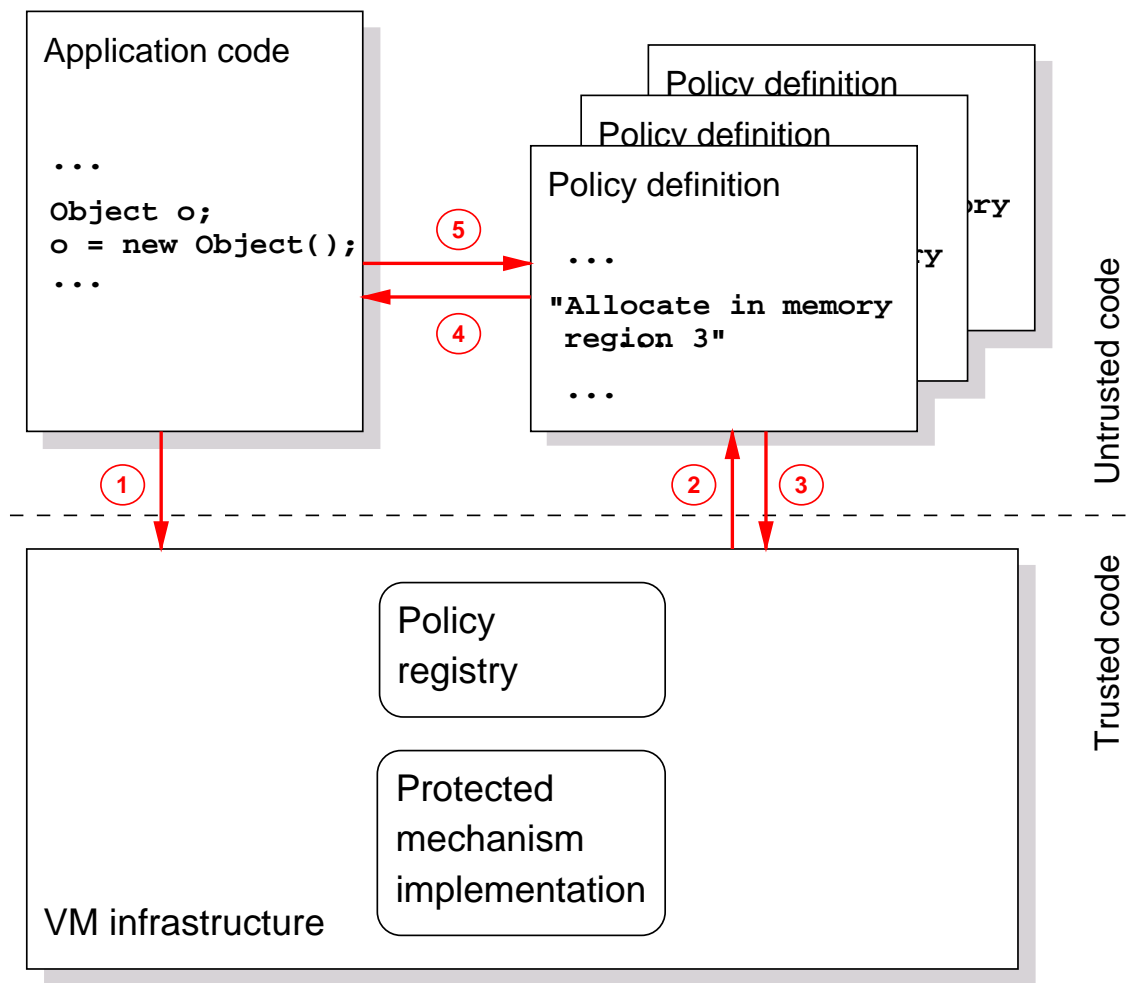
Mechanisms: provide first-fit, best-fit, segregated free lists, never-free, FIFO/LIFO allocation in a particular heap,

Policies: map allocation sites in the application to heaps

Problem domain: thread scheduling,

Mechanisms: storing and resumption of thread contexts, checking only runnable threads are resumed,

Policies: implemented by selecting which saved context to resume, e.g. round-robin, priority base, period-proportion, ...



Storage allocation

- As a running example, suppose that we want to segregate certain kinds of object – e.g. to improve spatial locality, avoid copying etc.

```
class Hashtable {  
    HashtableEntry entries[];  
}
```

← defined in a standard library and allocated by the application

```
class HashtableEntry {  
    Object key;  
    Object value;  
    HashtableEntry next;  
}
```

← allocated within Hashtable

Storage allocation (2)



- What do we need to do to be able to express useful policies?
- Deciding on a per class basis prevents us distinguishing different kinds of `Hashtable`
- Deciding on a per [static] allocation site basis still prevents us handling the `entries` array and instances of `HashtableEntry` differently between kinds of `Hashtable`
- Manually duplicating code into (e.g.) `LongLivedHashtable` and `LongLivedEntry` is ugly!

Isotopes



- Current solution is to introduce *isotopes*
- Objects of the same isotope are necessarily of the same class
- Different isotopes of the same class...
 - are indistinguishable to the language type-system and hence also to ordinary application code (and also the reflection API),
 - are implemented by the same *bytecode* methods,
 - but may be handled differently by policy implementations.

Example



```
public final class TransHeapAllocationUPI
    implements AllocationUPIIfc
{
    AllocationPMI heap =
        new VMDefaultAllocationPMI (10 * 1024 * 1024);

    AllocationPMI default =
        VMDefaultAllocationPMI.theVMDefaultAllocationPMI;

    AllocationIsotopeTag ait =
        new AllocationIsotopeTag ("applet objects");
}
```

```

public final AllocationPMI getAllocationPMI (Class c)
{
    StackFrameCursor    sfc;
    AllocationIsotopeTag sfait;

    sfc = AllocationContext.getAllocatingStackFrame ();
    sfait = sfc.getAllocationIsotopeTag ();
    if (sfait == ait) {
        AllocationContext.setAllocationIsotopeTag (ait);
        return heap;
    } else {
        return default;
    }
}
}

```

Thread scheduling

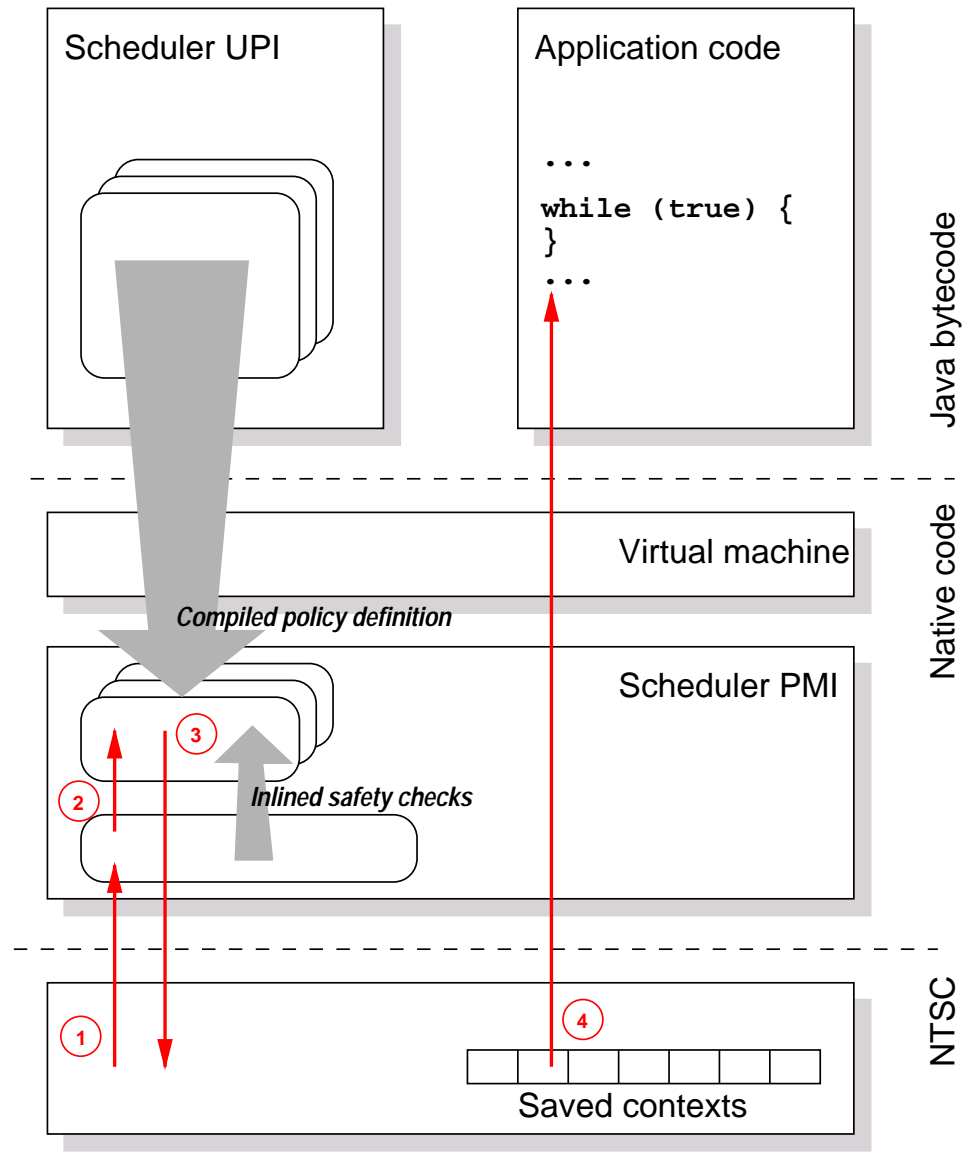


- Supporting untrusted thread scheduling policies presents alternative challenges
- Storage allocation decisions could be made once for each allocation site in each isotope – e.g. by in-lining into native code or rewriting bytecode
- Thread scheduling decisions may be needed much more frequently – the design must therefore allow an efficient implementation
- Similar concerns exist with safe user-level scheduling in operating systems

Thread scheduling (2)



- Based on *scheduler activations*
- The application-supplied scheduler receives an up-call whenever a decision is required, e.g.
 - a running thread blocks,
 - a blocked thread becomes runnable,
 - a new thread is created,
 - a thread's priority changes.
- It selects a context to resume
- This selection must be checked



Example

```
public void reschedule ()
{
    int j = (last_scheduled_thread + 1) % NUM_CONTEXTS;

    for (int i = 0; i < NUM_CONTEXTS; i ++) {
        if (state[j] == STATE_RUNNABLE) {
            last_scheduled_thread = j;
            NativeScheduler.resumeFromSlot (j);
        }
        j = (j + 1) % NUM_CONTEXTS;
    }
    NativeScheduler.blockProcess ();
}
```

Further work



- Implementation over current VMs: the existing implementation work is based on JDK 1.1.4 so performance comparisons are meaningless
- Extension to other policy domains, e.g.
 - garbage collection – to what extent is it possible for untrusted code to do more than select between existing collectors?
 - lock acquisition order
 - pre-fetching data access

Conclusions



- Many aspects of a VM implementation can be devolved to untrusted application code – the system described allows readable policy definitions and encourages separation of policies from application code
- Although not shown here, the interfaces support policies that benefit from run-time feedback – an ultimate goal is to allow the standard defaults to be implemented in this way

For more detailed references see

<http://www.cl.cam.ac.uk/~tlh20>