

# Bringing the full power of Java Technology to embedded realtime applications.

... How to make Java programs  
small, fast and deterministic.

Fridtjof Siebert, 16. April 2002

# Deeply embedded realtime applications



## Examples:

automotive, avionic, industrial automation, telecommunication, medical, ...

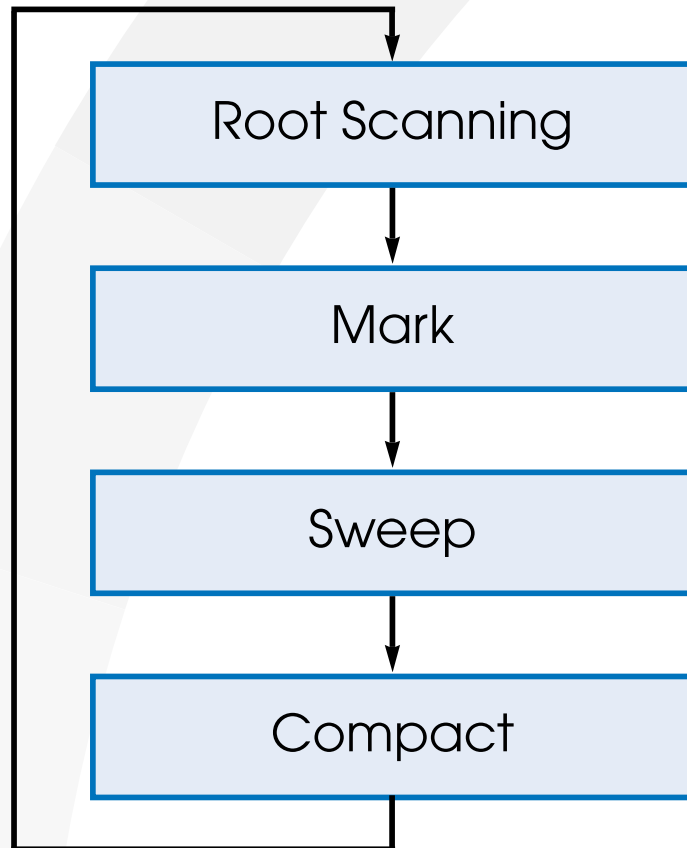
# Why Java for realtime systems?

- Higher productivity
- Plattform indepenence
- Reliability (type/pointer safe)
- Flexibility (dynamic loading)

## Problems:

- Memory requirements
- Poor runtime performance
- Lack of realtime guarantees

# Biggest Problem: Garbage Collection



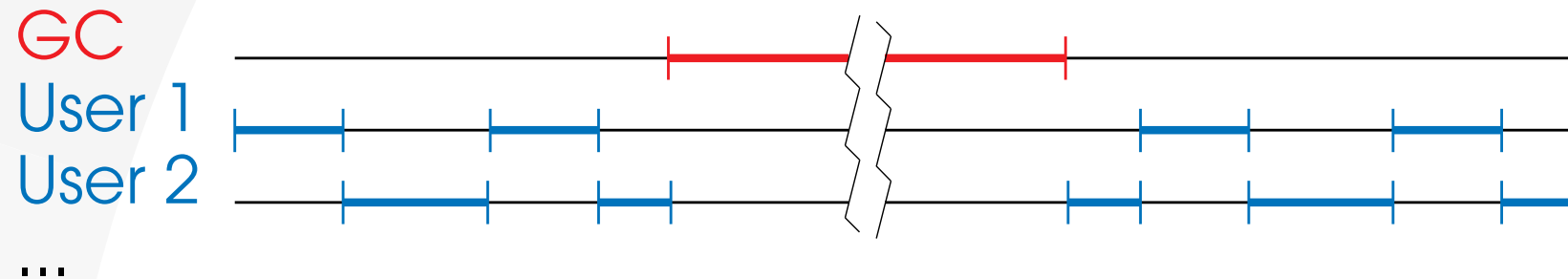
- Mark objects referenced from outside the heap
- Recursively mark all reachable objects
- Free memory of unmarked objects
- Move allocated memory to get contiguous free range

All of this happens while Java application is running!

# Classic Garbage Collection

GC can stop execution for long periods of time:

Thread: → time



## Problem:

long, unpredictable pauses during execution.

## 'real-time' Java extensions

e.g., the Real-Time Specification for Java (RTSJ),

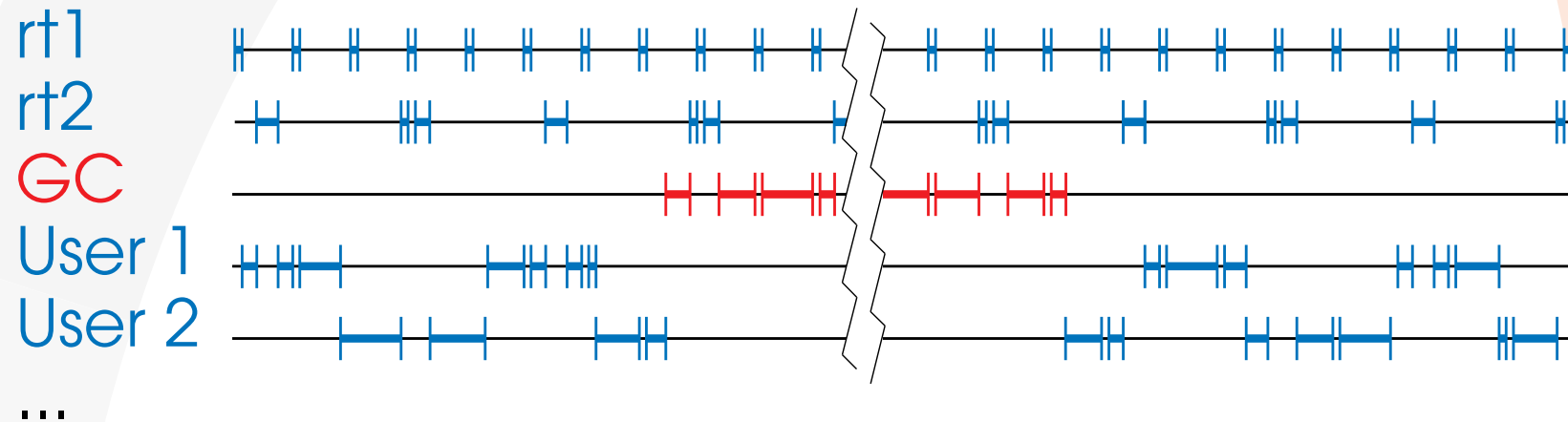
- define real-time Threads that can interrupt the garbage collector.
- restrict the Java functions allowed in these threads:
  - no allocation or use of special heaps that are not under the control of the GC
  - no or limited access to Java Heap
  - no synchronization with normal threads

# 'real-time' Java extensions

Support for 'real-time' threads:

Thread:

→ time



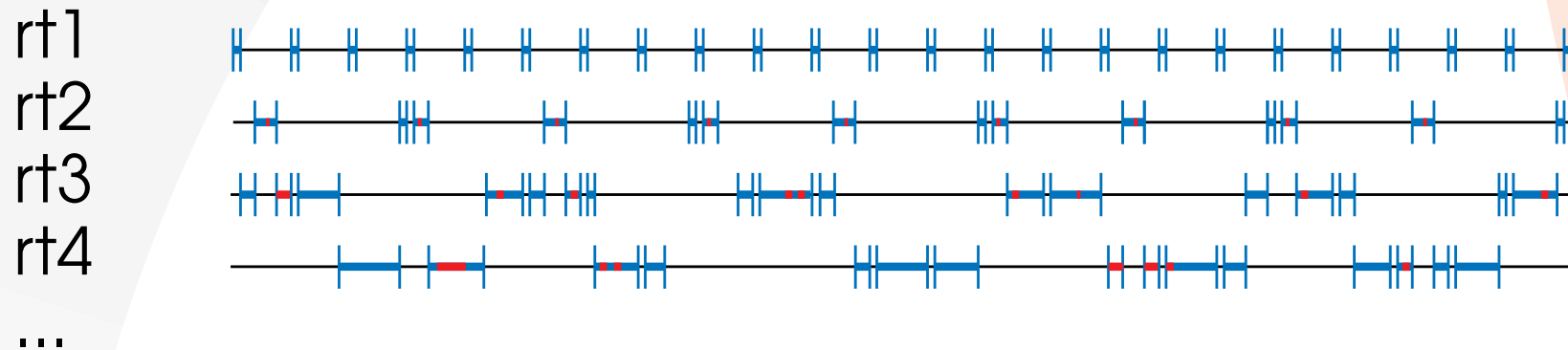
The application must be split into a realtime and non-realtime part. Synchronization between these parts is not possible!

# Realtime Garbage Collection

All Java threads are realtime threads:

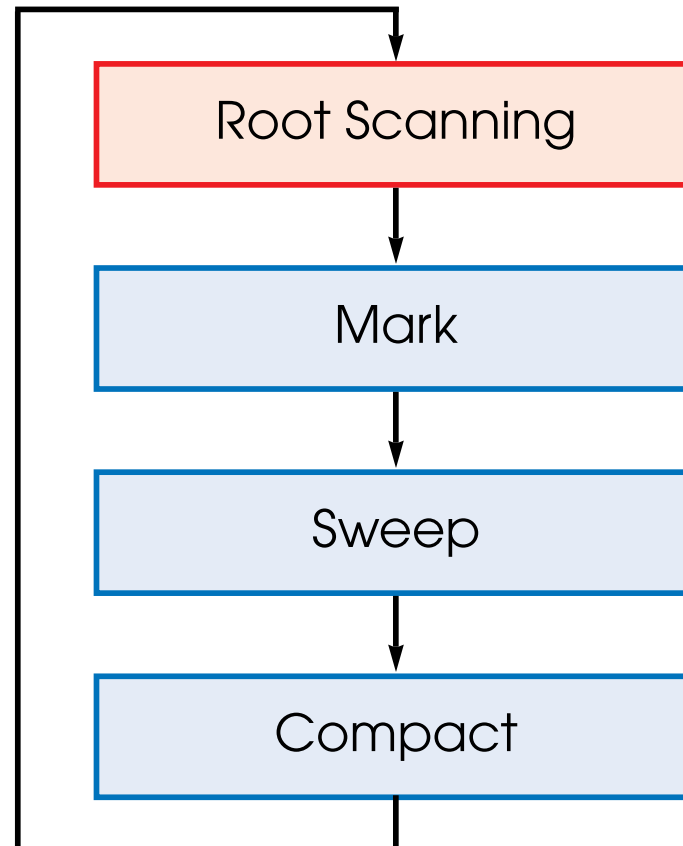
Thread:

—————→ time



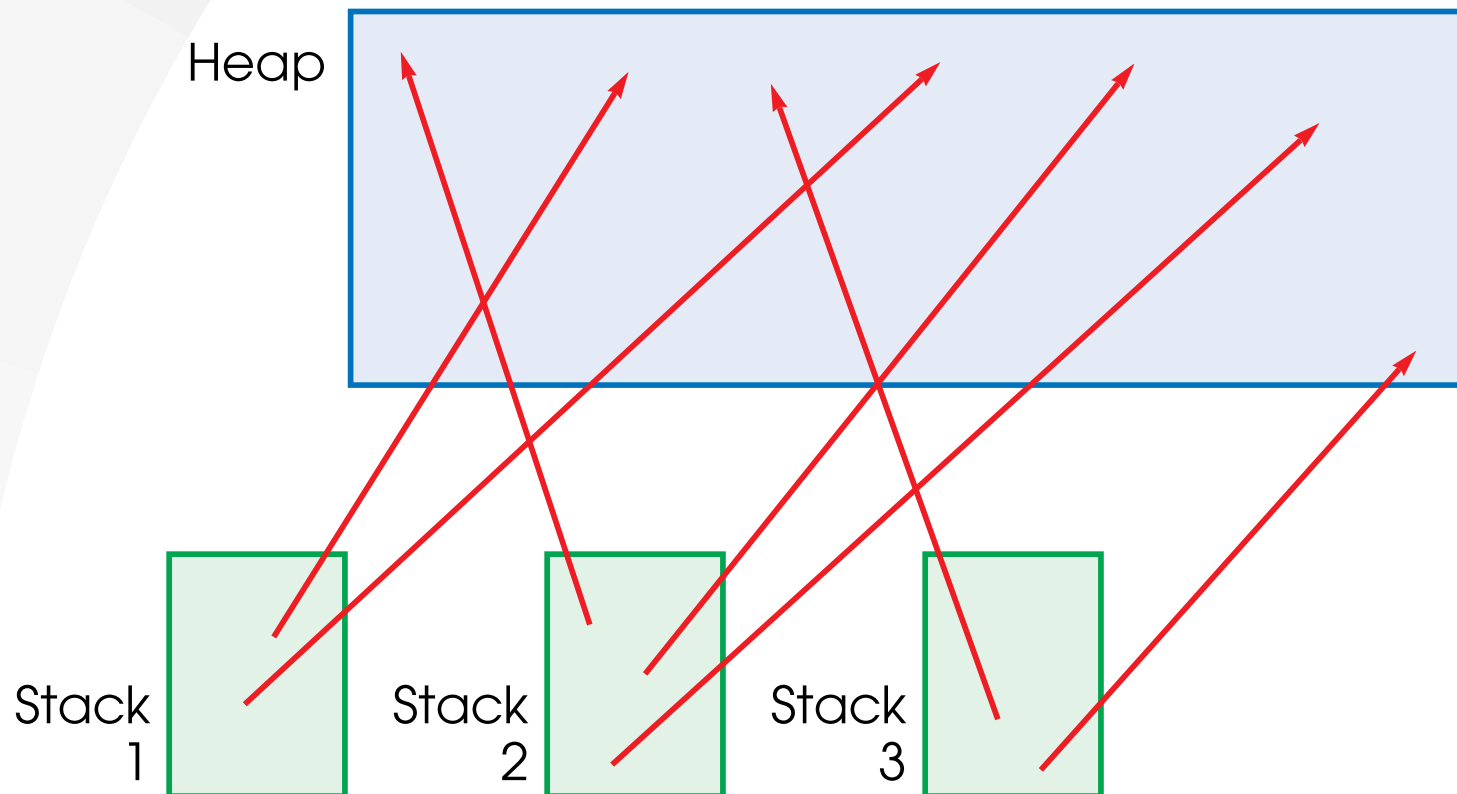
- All gc work is performed at allocation time.
- GC work must be sufficient to recycle enough memory before memory is exhausted.
- WCET for an allocation required.

# Root Scanning

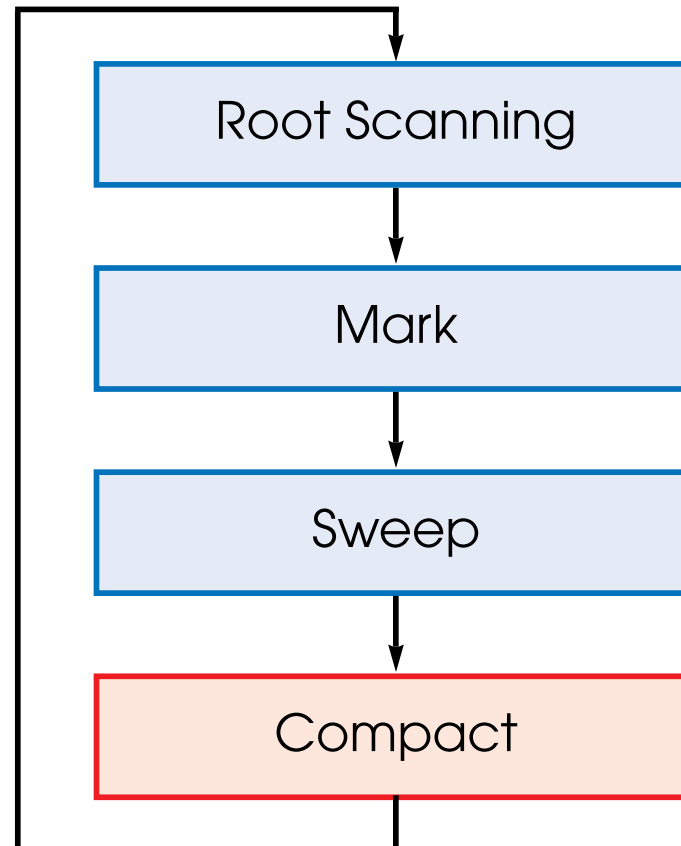


# Root Scanning

Stop threads to mark referenced objects.  
Or, stopping of thread through compiler support.



# Memory Compaction



# Memory Compaction

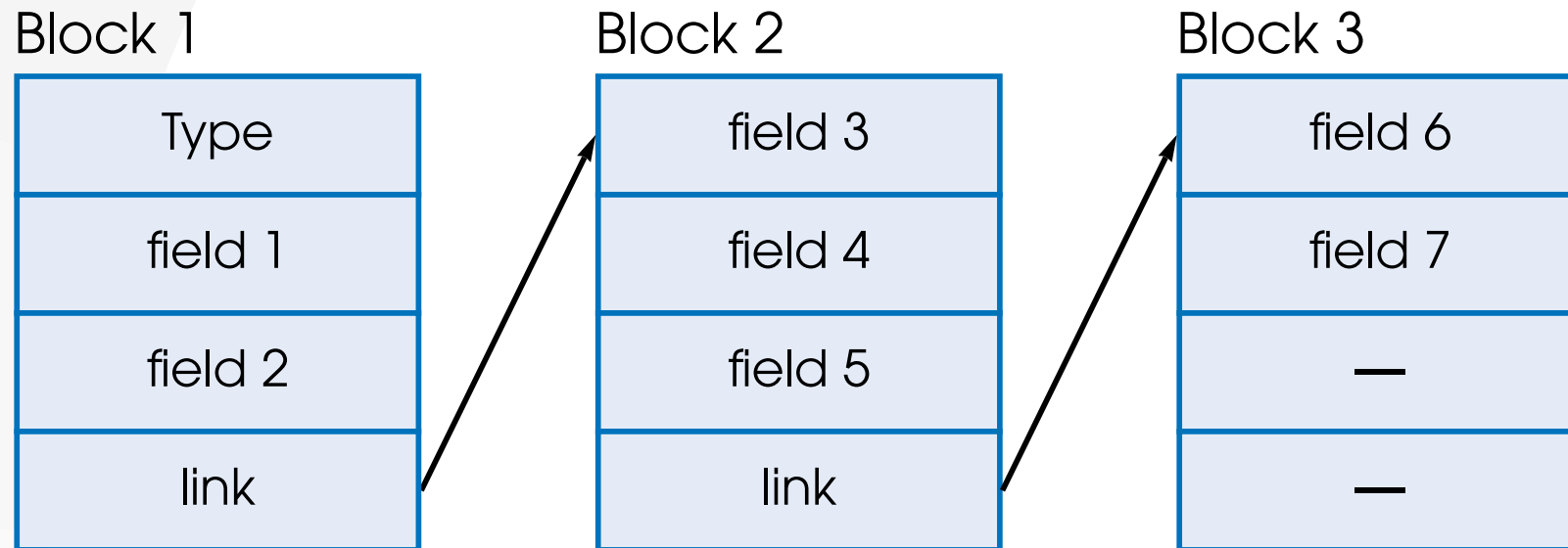
Possible Solutions in a realtime system

- Don't care about fragmentation  
not a solution for reliable applications.
- Do it to defragment the memory  
causes potentially long pauses.
- Move objects incrementally  
high object access overhead.
- Avoid fragmentation by using fixed-size blocks.  
avoids need for compaction completely.

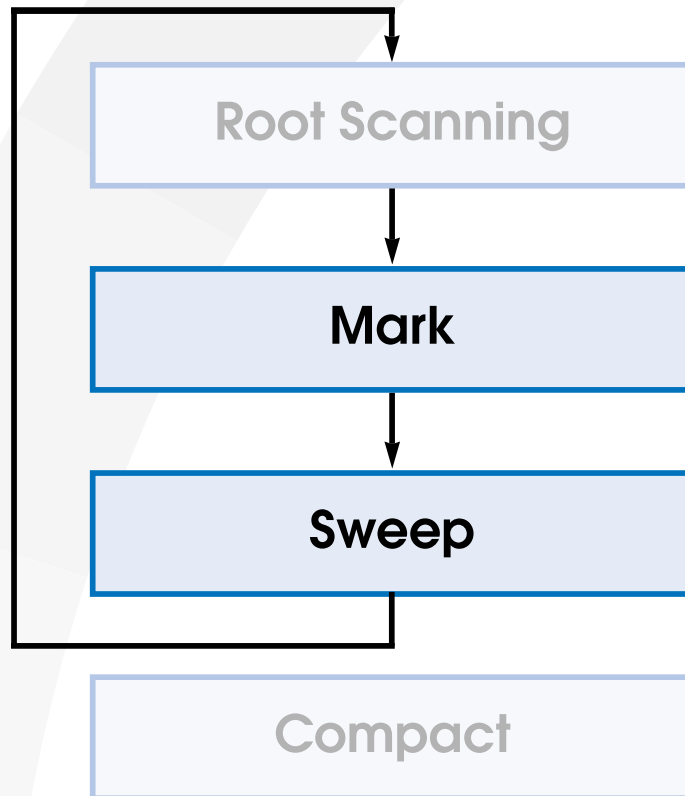
# Using fixed-size blocks

Regard heap as an array of fixed-size blocks.

- Build Java objects out of these blocks.
- No need to move objects for defragmentation



# GC Cycle in *JamaicaVM*



- Constant-time root scanning (compiler support)
- Incremental mark & sweep collector using very small increments of work
- Non-Fragmenting object layout: No compaction required!

# Example: GC configuration

Measure heap requirement of test application:

```
> jamaica -analyse 5 TestApp
> ./TestApp
[... out of TestApp...]
### Application used at most 117224 bytes for Java heap
###
### heapSize      wcet dynamic
### 397k          6
### 331k          7
### 240k          10
### 195k          14
### 182k          16
### 166k          20
### 150k          28
### 138k          40
### 122k          128
### 118k          256
```

## Example: GC configuration

Determination of worst-case execution time of

```
new StringBuffer()
```

First, determine number of blocks used by object:

```
> numblocks java.lang.StringBuffer  
1
```

Then, determine Worst-case execution time:

$$wcet = numblocks * max_{gc} * wcet_{gc\_unit}$$
$$wcet_{166k} = 1 * 20 * 2\mu s = 40\mu s$$
$$wcet_{331k} = 1 * 7 * 2\mu s = 14\mu s$$

# Further challenges for a real-time implementation of Java

- Dynamic calls (virtual calls, interface calls) require constant-time execution
- Type checks and casts require constant-time execution
- Class initialisation  
early initialisation would change semantics
- Synchronization (monitors)  
must be inlined for deterministic behaviour

# Reducing Memory Demand

Use of defined subsets of Java language and API as defined in configurations and profiles:

- CLDC (connected limited device configuration) subset of Java language (no long, double) minimal subset of Java API
- CDC (connected device configuration) subset of Java API e.g., no graphics (awt)
- MIDP Special APIs for mobile devices

# Reducing Memory Demand

## Classfile compaction:

Standard class files relatively large and not executable out-of-ROM.

- Use more compact format.
- Use format that is executable in place

ex. JEFF format defined by J-Consortium

This technique typically saves 50% of memory.

# Reducing Memory Demand

## Smart linking

Analyse application and remove unused code.

Together with compaction, this saves 80-90% of memory.

Example:

classes of emb. Caffeine: 334630 bytes

comp. + smart linking: 23280 bytes (-93%)

**But:** User attention required when dynamic class loading or reflection API is used!

# Reducing Memory Demand

## RAM requirements

- Execute bytecodes from ROM
- Small overhead for Java objects and arrays:
  - type information (class, array length)
  - hash code
  - monitor
- Small GC overhead
  - location of references
  - GC state of object
  - finalization state of object

# Execution Speed

**Interpreter is often too slow, so use of compilation is required, but**

- Just-in-Time-Compilation not deterministic and not applicable in realtime systems
- Flash-Compilation (at load time) deterministic, but requires
  - memory for compiler on target
  - memory for compiled code
- Ahead-of-Time compilation is an alternative

Typical speedup through compiler: 10-30 times!

# Ahead-of-Time compilation

Compile at build time:

- No compilation on target system to avoid overhead (memory, time) and unpredictability of JIT-compilation.
- Need to allow mixing of compiled and interpreted code. This enables
  - small footprint and
  - dynamic loading.

# Profile guided compilation

Compiled code requires significantly more memory than interpreted bytecode.

Compiling a small percentage is sufficient:

<b>compiled</b>	<b>caffeine score</b>	<b>speedup</b>	<b>code size</b>
0%	202	1.0	174 784
1%	1399	6.9	188 384
2%	3533	17.5	190 592
5%	6728	33.3	194 176
10%	7751	38.4	202 368
20%	7546	37.4	221 376
50%	7779	38.5	258 816
100%	7788	38.6	268 512
all	7792	38.6	305 760

# Integration of low-level features

Need to access hardware, system functions, etc.

- **JNI** — Standard Java Native Interface  
Slow and big, but portable.
- Proprietary interfaces —  
can be optimized for efficient access.

Alternative

- Direct hardware access in Java through special APIs.  
ex: RTSJ's PhysicalMemory  
RTDA (J-Consortium)

## Conclusion

A Java implementation for embedded and realtime systems can provide solutions to make Java deterministic, small and efficient.

The developer of these systems can profit from the advantages that made Java technology so popular.



Try it! Free evaluation  
at [www.aicas.com](http://www.aicas.com).

More information?

send e-mail: [siebert@aicas.com](mailto:siebert@aicas.com).