JamaicaVM — Java for Embedded Realtime Systems

... bringing modern software development methods to safety critical realtime applications

Fridtjof Siebert, 25. Oktober 2001

Deeply embedded realtime applications





Examples:

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

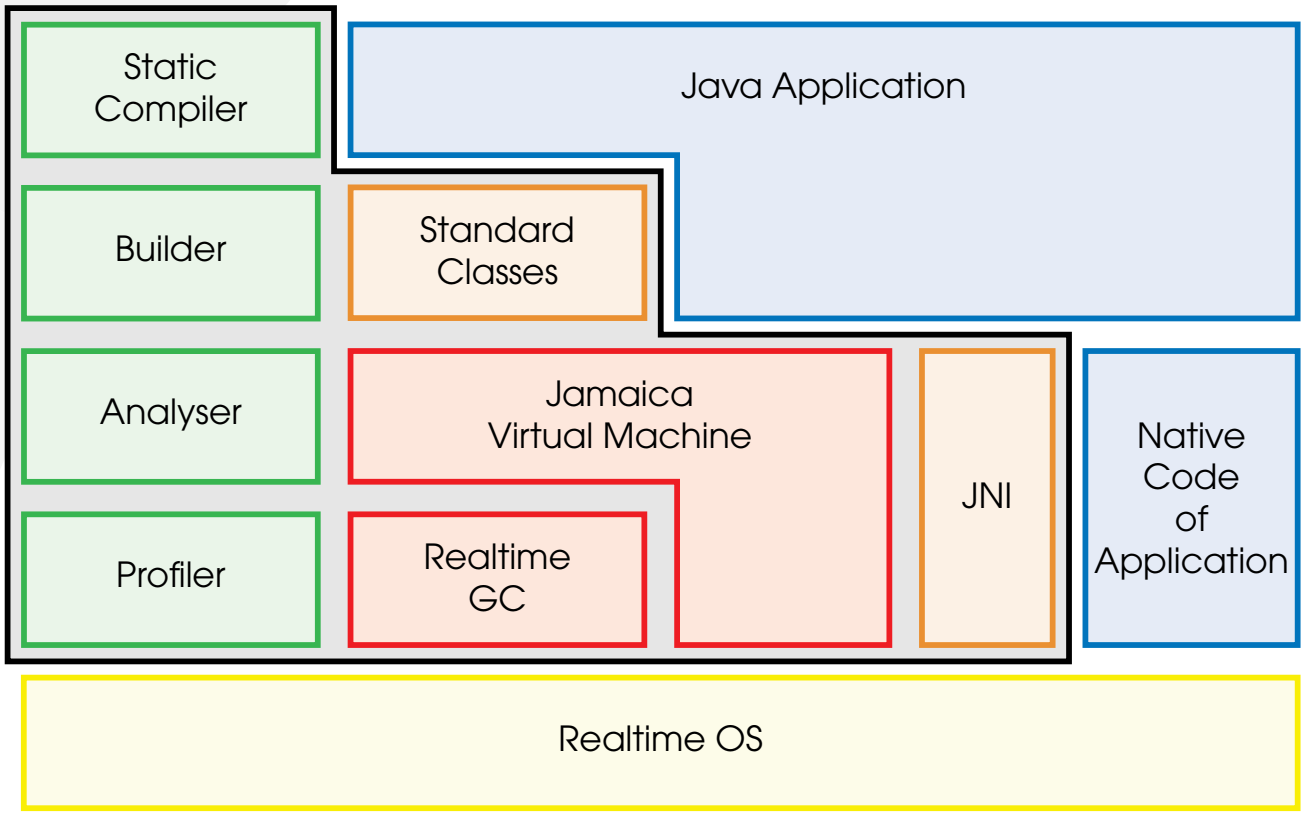
Why Java for realtime systems?

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

Problems:

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

The *JamaicaVM*



JamaicaVM

JamaicaVM

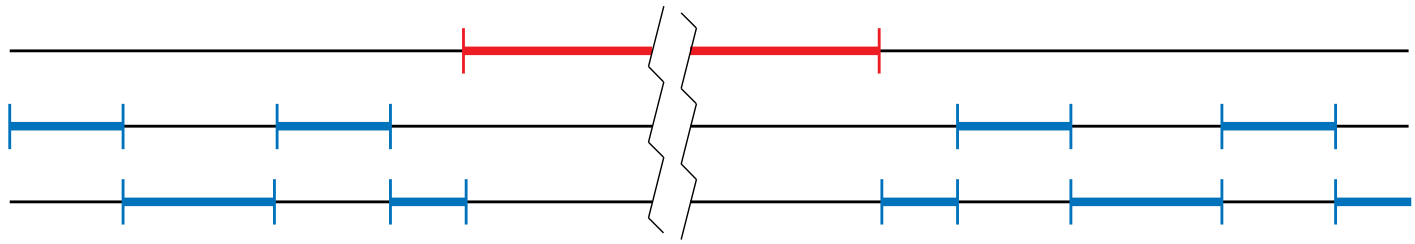
- Java-Implementatation based on JDK 1.2
- Exact realtime garbage collector without GC interruptions and guaranteed allocation time of a few µs.
- Builder Tool to create small and efficient applicats for embedded systems.
- static, optimizing compiler for maximum performance
- Java Native Interface to access existing code

Classic Garbage Collection

GC can stop execution for long periods of time:

Thread: ⟶ time

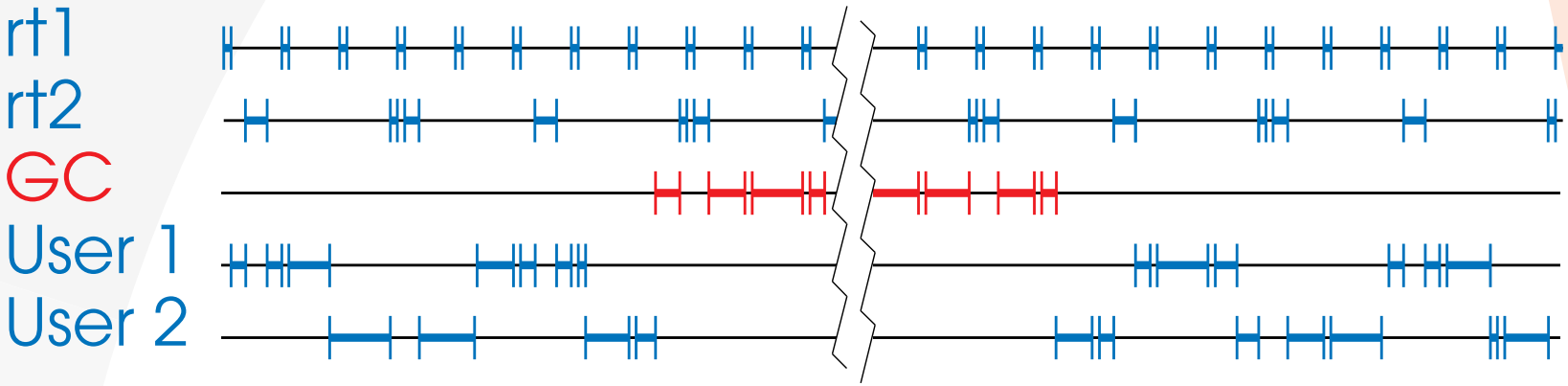
GC
User 1
User 2
...



'real-time' Java extensions

Support for 'real-time' threads:

Thread: ⟶ time



...

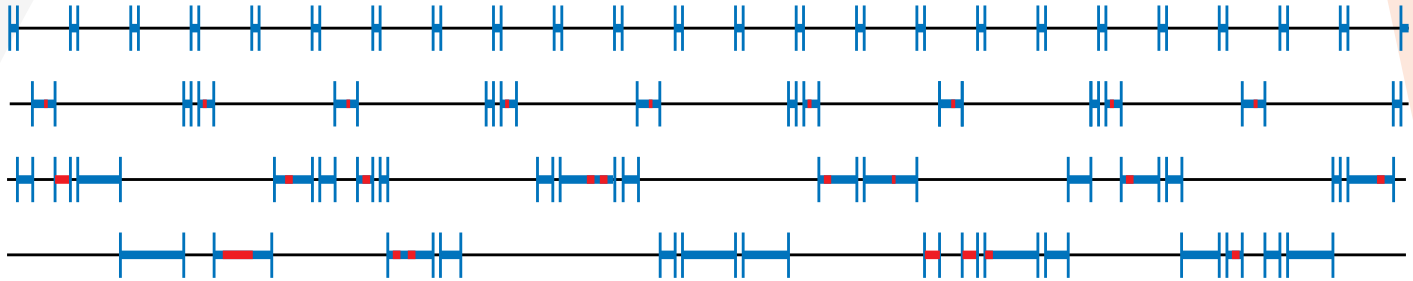
The application must be split into a realtime and non-realtime part. Synchronization between these parts is not possible!

JamaicaVM: realtime threads

All Java threads are realtime threads:

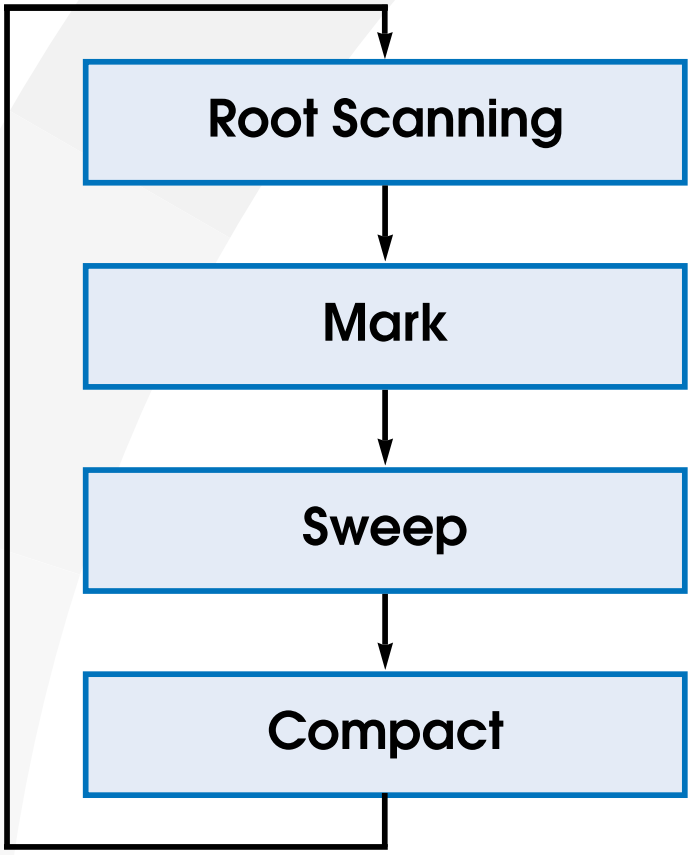
Thread: ⟶ time

rt1
rt2
rt3
rt4
...



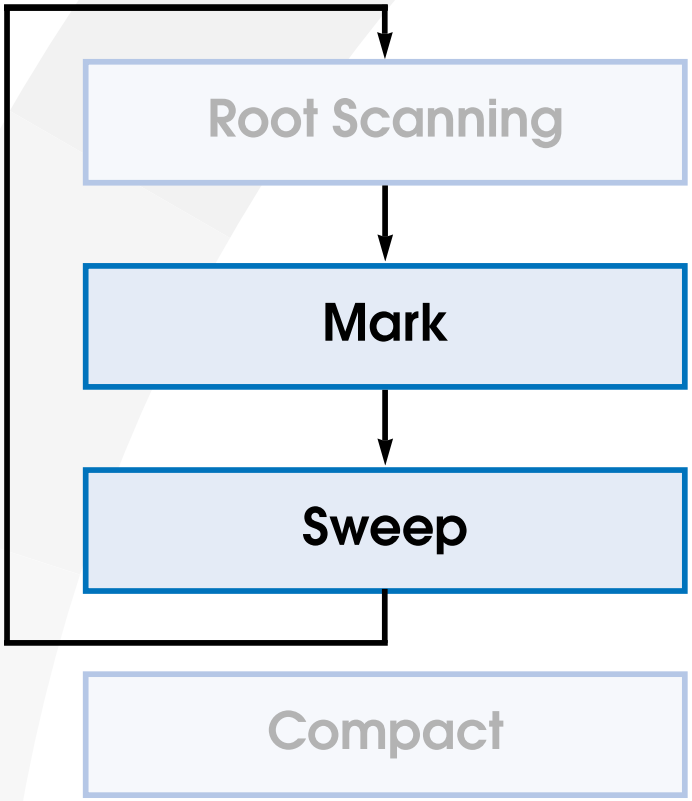
All garbage collection work is performed at allocation time. GC work is sufficient to finish GC cycle before memory is exhausted. WCET for an allocation can be given.

Typical GC cycle



- 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

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 * 2µs = 40µs

$wcet_{331k}$ = 1 * 7 * 2µs = 14µs

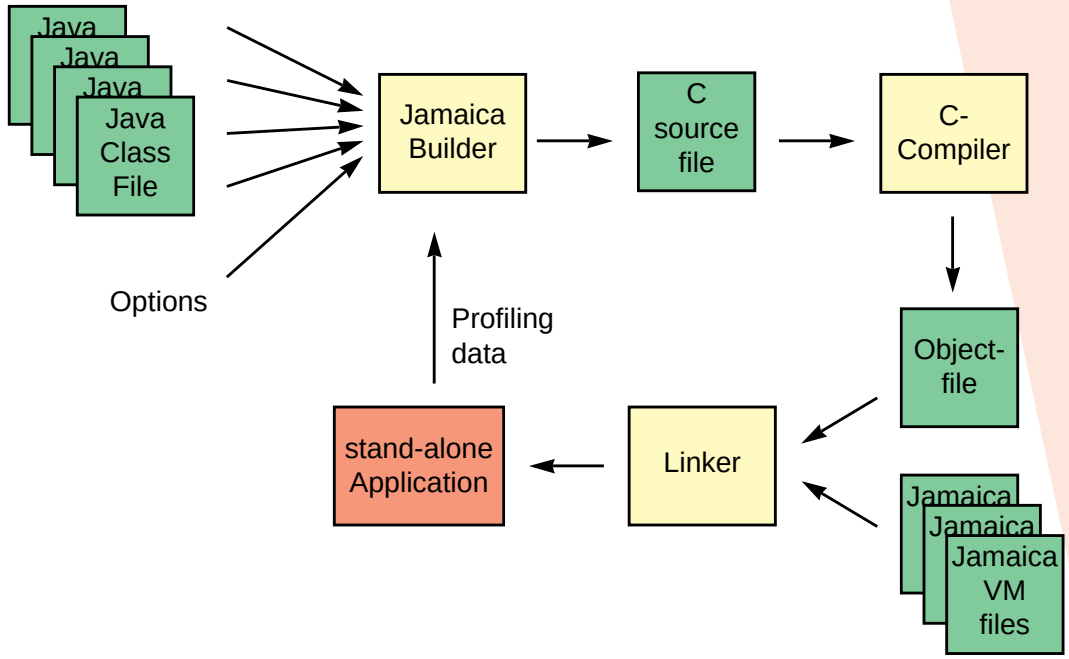
Further challanges for a real-time implementation of Java

- 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

The *Jamaica* Builder

Create single application out of Class files and ***JamaicaVM***:

- compaction
- smart linking
- compilation
- analysis
- profiling
- ...



Reducing Memory Demand

Classfile compaction:

Avoid redundancies in the Java classfile format.

Saves typically 50% of memory.

Example:

classes of emb. Caffeine:	334630 bytes
after compaction:	149268 bytes (-55%)

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

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

JamaicaVM: Optimizing Static Compiler

Compile at build time:

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

Typical speedup factor 20-25.

Profile guided compilation

Compiled code requires significantly more memory than interpreted bytecode.

Compiling a small percentage is sufficient:.

compiled	caffeine score	speedup	code size
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 native code

Often, C/C++ needs to be accessed to

- Use system features
- Access library code
- Integrate legacy code

JamaicaVM provides two means to perform this

- JNI — Standard Java Native Interface
  Portable betweent differen VMs
- JBI — Jamaica Binary Interface
  Optimized for efficient access.

Conclusion

Java can be used even in deeply embedded applications with limited resources and need for realtime behaviour.

Object-oriented Java programming using dynamic allocation is possible even for time-critical code.

This brings higher productivity, reliability, deterministic execution and easy code resuse and capsulation.