J2ME Best Practices

'

'

'

'

'

Java ME is becoming more sophisticated
● Very different from KVM-based stacks!
● Hardware acceleration, dynamic compilation,
runtime optimizations, advanced garbage collection,
3D graphics support, more...
● Tuned for certain code patterns and common constructs
● But breadth and depth of optimizations is limited
● Analysis and optimization constrained in space
and time
● Can't do “magic”
● Understanding platform properties is critical

● Increase the chances your code will run faster
● No guarantees: By definition can't target particular
implementations or specific features
● Focus on Java VM-level optimizations
● Disclaimer
● There are many different Java ME platforms out there!
We don't know them all. We haven't tried them all
● We can only make educated guesses about their
internal design and their properties
● Quantitative statements can be very difficult
● Your mileage will vary

'

Application startup
● Delay until user can interact with application
● One-time initialization and startup costs
● Long startup (>2 sec) is a major irritant
● Caused by jar loading, class loading/cp resolution, initialization, object creation, resource handling, UI setup

'*Application execution speed and consistency*
● ***Continuous low overall execution speed***
● ***Long computation times get annoying***
● ***Caused by lack of runtime optimizations, repeated failure to optimize, inefficient overall execution***
● ***Repeated changes between fast and slow execution***
● ***Inconsistent performance is hard to cope with***
● ***Caused by shifting memory use patterns, garbage collection, undoing of existing optimizations due to resource constraints***
'

Application liveliness
● Repeated pauses and “hangs” during execution
● Feedback delays and pauses >50 ms are noticeable
● Caused by costly runtime optimization, code transformations, garbage collection

“Proficient Design”
● The basics of good application design and efficient code
“Java VM-Friendly Design”
● Target properties and techniques of advanced Java ME platforms

Big rules
● Choose an efficient algorithm
● A mismatched algorithm will never be efficient
● Reduce application startup time
● Interact with user as quickly as possible
● Do less work initially, initialize lazily or in background

(C) Copyright Nokia 2013. All rights reserved.

- Note: Loading of resources, images particularly heavyweight
- Maintain responsiveness of UI
- Use threads for blocking or long-running operations
- Provide feedback to the user on what's happening

Efficient programming
- Use scalar types, local (i.e. stack) variables
- Pull invariants and string literals out of loops
- Avoid String.append() and String.equals()
- Use Exceptions only outside performance-critical code
- Release objects by setting references to them to null
- Use arrays instead of collections (Vector, HashTable)
- Cache results and objects locally where it makes sense
- Replace multiply and divide operations with shift
- Use best suited (read: efficient) library calls

'

More efficient programming
- Use Inner classes judiciously (extra space/time cost)
- Compare against 0, null, or false
- Use int instead of long where possible
- Use floating point carefully, as it is often emulated
- Avoid large or unnecessary synchronization blocks
- Avoid unnecessary object allocations
- Don't initialize class members to default value
- Try to avoid gc where visible to the user
- Obfuscate your code for size reduction

Advanced Java VM Techniques
- Hardware acceleration
- Static-/Ahead-of-time compilation (AOT)
- Just-in-time compilation (ADC)
- ...or a combination of the above
- Garbage collection techniques
- Startup improvement techniques
- Often very platform-specific, not discussed
- Graphics- and UI-subsystem techniques
- Often external to Java VM, dominated by native code or hardware/graphics architecture. Not discussed

Initialization

1) Static Array Initialization
- Problem
- Array initialization performed in static initializer
- This can cause significant code bloat and slow down class loading and initialization
- Impact
- Application startup, memory footprint
- Guideline
- Move initialization out of static initializer and do programmatic initialization or read data from I/O
- Note: Time vs. space trade-off (I/O can be slow)

Runtime Optimizations

2) Locality of Hot Code

- Problem
- Hot code spread out over multiple individual methods
- Optimization techniques typically operate on method boundaries (for semantical reasons)
- This multiplies overhead for hot spot detection, optimization, and subsequent management
- Impact
- Liveliness, execution consistency
- Guideline
- Factor/concentrate hot code in a few methods

3) Large Methods Containing Hot Code
- Problem
- Hot code embedded in large methods
- Optimization typically operates with method granularity
- Substantial hit taken for optimization of large body of code, and possible resource constraints or optimization failure (complexity or lack of resources)
- Impact
- Liveliness, execution consistency
- Guideline
- Keep methods with hot code compact

4) Mixing Hot and Cold Code

(C) Copyright Nokia 2013. All rights reserved.

- Problem
- A section of code is hot but contains a number of code paths that don't execute often
- This results in large methods (see previous slide) and disrupts optimization and execution due to branches
- Impact - Execution speed
- Guideline -- Avoid mixing hot and cold code - all code in a hot method should really be hot

' ''''5) Conditional Exceptions in Hot Code

- Problem
- Code conditionally throws exceptions in the hot path
- This precludes certain optimizations (code reordering/rescheduling); exceptions are expensive
- Impact
- Execution speed
- Guideline
- Never throw unconditional exceptions in hot code
- Avoid conditional exceptions if possible
- Caveat: Might not always be practical due to implicit exception points such as null checks, array bounds, etc.

'***6)*** ''''System.gc()

- **Problem**
- **Application periodically calls System.gc()**
- **Sophisticated memory management systems already dynamically adapt to a variety of conditions**
- **This means calls to System.gc()likely add overhead without any benefit (or even cause disruption)**
- **Impact - Execution speed**
- **Guideline - Don't call System.gc()**

Use Device Resources Wisely

• Minimize application size
– Limit the number of features
– Object design – create single objects for common tasks and reuse them
– Place resource data on the server
'

' ''''Minimize runtime memory use

– Use scalar types (int, boolean) instead of objects (String, Integer)
– Don’t depend entirely on the garbage collector
  • Avoid creating a lot of objects quickly
  • Reuse objects
– Move computation to server
  • Perform data filtering, sorting, etc. on the server before sending it to the device
  • Completely process and manipulate data as needed before indicating to the user that new data has arrived. This hides latency from user

'***Code*** '''' Performance

• Use local variables
– Generally quicker to access local variables than class members
• Use threads for lengthy operations (> than 1/10 second)
– Stop the operation from blocking the main UI thread
• Set appropriate access and declaration type
– Declare members as private whenever possible
– Declare classes as final if they will not be extended
– Carefully use static and final – understand the performance impact on both class and primitive types
• Carefully optimize loops and mathematical operations

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.