Monitors and Exceptions: How to Implement Java efficiently

Andreas Krall and Mark Probst
Technische Universitaet Wien
Outline

• Exceptions in CACAO
  – Exception implementation techniques
  – CACAO’s implementation
  – conclusions

• Monitor Implementations
  – SUN ’s monitors
  – CACAO
  – “Thin Locks”
  – Meta-locks
  – conclusions
Exceptions in Java

- Implicit: null pointer, array out of bounds, division by 0
- Explicit (throw)

- Catching an exception
  
  ```java
  try { … }
  catch (Exception_1 e1) { … }
  ...
  catch (Exception_n en) { … }
  finally {…}
  ```
Exceptions in Java (cont.)

```java
class E {
    public void f() {
        int i;
        try {
            try {
                i = 1/0;
            }
            // catch (ArithmeticException ae) {
            //     ae.printStackTrace();
            // }
            catch (ArithmeticException ae) {
                ae.printStackTrace();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
        // catch (Exception e) {
        //     e.printStackTrace();
        // }
    }
    Method void f()
    0  iconst_0
    1  istore_1
    2  iconst_1
    3  iconst_0
    4  idiv
    5  istore_1
    6  goto 22
    9  astore_2
   10  aload_2
   11  invokevirtual #7 <Method void printStackTrace()>
   14  goto 22
   17  astore_2
   18  aload_2
   19  invokevirtual #7 <Method void printStackTrace()>
   22  return

Exception table:
    from  to  target  type
    2     6     9  <Class java.lang.ArithmeticException>
    2     17    17  <Class java.lang.Exception>
```
class EF
{
public void f()
{
    int i=0;
    try {
        i=1/0;
    }
    catch (ArithmeticException ae) {
        ae.printStackTrace();
    } 
    finally {
        i++;
    }
}}

Method void f()
0 iconst_0
1 istore_1
2 iconst_1
3 iconst_0
4 idiv
5 istore_1
6 goto 19
9 astore 4
11 aload 4
13 invokevirtual #6 <Method void printStackTrace()>
16 goto 19
19 jsr 31
22 goto 37
25 astore_2
26 jsr 31
29 aload_2
30 athrow
31 astore_3
32 iinc 1 1
35 ret 3
37 return

Exception table:
from   to  target type
 2     6     9   <Class java.lang.ArithmeticException>
 2    19    25   any
Exceptions in Java (cont.)

• Each method has an exception table
• An entry in the table contains
  – Address of the exception handler
  – bytecode address range for which the handler is used
• When an exception occurs
  – If it is caught, the handler is executed
  – If it is not caught, it is thrown to the calling method
• Code motion limitations
  – Before the exception raising instruction all code must have been executed
  – No instruction after the raising instruction can be started
Exception implementation techniques

• Static try block table (Java)
  – Check and search the exception at run time

• Dynamically create a list of try block data structures (C++, Ada)
  – Drawback: creates a data structure even if the exception is not thrown (the common case)

• Function with 2 return values (old CACAO)
  – An additional register is set to non-zero if an exception is thrown and not caught, and the function returns
  – At function return if register is non-zero, the handler is executed
Motivation for a change

- # of method invocations - 2 magnitudes bigger that # of try blocks
- Exceptions are rarely raised
- Lots of null pointer checks (in Java at run time)

<table>
<thead>
<tr>
<th></th>
<th>JavaLex</th>
<th>Javac</th>
<th>Espresso</th>
<th>Toba</th>
<th>Java_cup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null pointer checks</td>
<td>6859</td>
<td>8197</td>
<td>11114</td>
<td>5825</td>
<td>7406</td>
</tr>
<tr>
<td>Method calls</td>
<td>3226</td>
<td>7498</td>
<td>7515</td>
<td>4401</td>
<td>5310</td>
</tr>
<tr>
<td>Try block</td>
<td>20</td>
<td>113</td>
<td>44</td>
<td>28</td>
<td>27</td>
</tr>
</tbody>
</table>
The new exception handling scheme

• CACAO
  – JIT
  – Fastest JVM for Alpha processor (1998)
• Goal : generate native code by CACAO JIT
• Achieved
  – Reduced generated code by a half (compared with the old CACAO)
  – Run-time check of null pointers done by hardware
CACA O stack frame

• Contains only copies of
  – Saved registers
  – Spilled registers

• Doesn’t contain
  – The saved frame pointer
  – Size of the frame (used only by frame allocating/de-allocating routines)

• Additional information is needed for exception handling
CACA O exception handling

• Method layout in CACA O
  – Constants
    • framesize
    • isleaf - flag which is true if the method is a leaf
    • intsave - # of saved integer registers
    • floatsave - # of saved FP registers
    • extable - exeption table ( similar to JVM table)
  – Code
CACAO exception handling

• Mechanism: similar as Java, but at native code level
  – Check if there is a handler for the raised exception
  – Yes: run it
  – No: unwind the stack and search in the parent
    • Info from constant area is used for register restoration and stack pointer update

• Bytecode must be translated in native code: complications
  – Elimination of “dead” basic blocks: info about them must be kept if the exception table points to it
  – No reordering of basic blocks allowed (?)

CACAOl exception handling (cont)

• No explicit null pointer checks
  – First 64K of memory protected against r/w
  – If a segmentation violation occurs
    • catch the signal
    • if within 64k generate null pointer exception
Results and conclusions

- Exception handling scheme in CACAO
- Not noticeable improvement in the run-time (3 %, but inaccuracy of measurement in the same range)
- Code size nearly halved

<table>
<thead>
<tr>
<th>CACAO old</th>
<th>61629</th>
<th>156907</th>
<th>122951</th>
<th>67602</th>
<th>87489</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACAO new</td>
<td>37523</td>
<td>86346</td>
<td>69212</td>
<td>41315</td>
<td>52386</td>
</tr>
</tbody>
</table>
Monitor implementations

- SUN’s monitors
- CACAO
- “Thin Locks”
- Meta-locks
- conclusions
Synchronization constructs in Java

• Synchronized methods
  – When executed, the thread tries to lock the object
  – If object not already locked by other thread, it succeeds and executes the method body
  – If another thread holds the lock the current thread blocks until the lock is released

• “synchronized” statement
  synchronized (expr) {
    statements
  }
  – Same rules as for synchronized methods
Java monitors versus “classical” monitors

• Java monitors are transparently embedded into the object (any object is a monitor)
• Java monitors may be entered recursively by the same thread
• Java monitors can use only a single implicit condition variable (wait/notify mechanism)
Wait/notify/notifyAll

- All can be called only in a synchronized method or in a synchronized statement
- `wait()` - blocks the current thread until a notification is sent
  
  ```
  synchronized (o) { …
      while (!condition) wait(); …}
  ```
- `notifyAll` - notifies all the waiting threads that the condition has changed
  
  ```
  synchronized (o) { …
      change condition ; notifyAll();}
  ```
- `notify` - notifies only one waiting thread
Bytecode representation of synchronization

• Bytecode instructions: monitorenter and monitorexit
• Synchronized methods
  – Don’t use monitorenter and monitorexit
  – Each method has a flag ACC_SYNCHRONIZED, which is set if the method is declared synchronized
    – If flag set, current thread tries to acquire the lock first
• “synchronized” statements
  – Use monitorenter and monitorexit
Sun’s monitor implementation

- **Object table**
  - Entries called *handles*: heap reference to an object, therefore unique (object identifiers)

- **Monitor cache**
  - Table which maps a handle to a monitor structure

- **Monitor structure**: data for performing the synchronization

- Whenever a thread synchronizes on an object, it first checks if the handle is mapped to a monitor structure
  - A table lookup must be performed
  - A monitor structure is created if necessary
Sun’s monitor implementation

- **Space**
  - Space efficient: monitor structures created only when threads try to synchronize on objects

- **Time**
  - Not efficient: a table lookup must be performed for each synchronization

- **Scalability**
  - Not scalable: monitor cache is a point of contention between threads
Alternative monitor implementations

• David Bacon & comp : Thin locks: Featherweight Synchronization for Java (IBM T.J. Watson RC), PLDI ‘98
• Andreas Krall & Max Probst : Monitors and Exceptions : How to Implement Java Efficiently, Java Workshop for HP Computing ‘98
• Ole Agesen & comp: An Efficient Meta-lock for Implementing Synchronization (Sun), OOPSLA ‘99
Common cases (Bacon &comp)

• locking an unlocked object
• locking an object already locked by the same thread a small number of times
• locking an object already locked by the same thread many number of times
• attempting to lock an object already locked by another thread, for which no other threads are waiting
• attempting to lock an object already locked by another thread, for which other threads are waiting
CACAO monitors

- monitorenter and monitorexit implemented using mutexes
- Observation: number of mutexes locked in the same time is small
- Use a mutex cache: implemented as a hash-table
- First entry in the bucket never de-allocated (most frequent case uses it w/o incurring allocate/deallocate costs)
CACA
do

• Space
  – Very efficient: worst case # of mutexes = # of buckets + # of parallel mutexes

• Time
  – Hash table lookup is fast (especially for a small # of mutexes)
  – Allocation/deallocation time spent in the most common case

• Scalability
  – Hash-table of mutexes - still contention point
Thin locks

- Used for first 2 common cases
- If any other case occurs, the lock is “inflated” and never “deflated” again
- Use 24 bits in the object header (if already available: no space overhead!)
  - 1 bit: thin/fat lock
  - 15 bits: owning thread
  - 8 bits: nesting count
- When a thread acquires the lock it becomes the owner of it (by using a compare-and-swap operation)
- When it releases the lock, it restores the ownership to 0
Thin locks (cont)

- Only the owner manipulates the synchronization data (different in the Meta-locks case)
- Inflation: the thread owner field is converted into a pointer to a data structure which contains:
  - Thread owner
  - Nesting count
  - Queue of waiters
- If the thread $t_1$ holds a thin lock and the thread $t_2$ tries to access it, $t_2$ will
  - Spin-lock until $t_1$ releases the lock (bad!)
  - Inflating the lock afterwards
Thin locks (cont.)

• **Space**
  – If 24 bits available in object header: no space overhead for the common cases!
  – Still efficient for the uncommon cases: space needed only when synchronization is performed

• **Time**
  – Very efficient in the common cases (no lookup needed, synchronization data locally available)
  – Problems can occur with spin-locking

• **Scalability**
  – Scalable: synchronization information kept by each owning thread
Meta-locks

- Two level scheme for synchronization
- *meta-locks* protect the access to the synchronization data (any thread can modify it)
- Only 2 bits in the object header are needed
- The other 30 bits of the word are displaced into a data structure which contains synchronization data.
- When a thread tries to perform a synchronization operation, it first acquires the meta-lock
  - If no other thread has the lock, it acquires it and releases the meta-lock
  - If some other thread has the lock, the thread adds a record to the queue of waiters and then releases the meta-lock
Meta-locks (cont)

• When a thread tries to perform a synchronization operation, it first acquires the meta-lock (quick if no contention)
  – Acquiring the lock
    • If no other thread has the lock, it acquires it and releases the meta-lock
    • If some other thread has the lock, the thread adds a record to the queue of waiters and then releases the meta-lock
  – Releasing the lock
    • If no other threads are trying to acquire the lock it just releases the metalock
    • If other threads are waiting in the queue, it wakes up the next in the queue
Meta-locks (cont)

• Space
  – Only 2 bits per object are needed for objects that never synchronize (thin-lock 24 bits regardless)
  – Amounts to total size of lock records (small compared to the necessary heap & stack space)

• Time
  – Very efficient (no lookup needed)
  – No spin-lock as thin locks

• Scalability
  – Scalable (no centralized contention point)
## Conclusions

<table>
<thead>
<tr>
<th></th>
<th>Space efficiency</th>
<th>Time efficiency</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sun</strong></td>
<td>-efficient</td>
<td>-inefficient</td>
<td>-not scalable: monitor cache is contention point</td>
</tr>
<tr>
<td></td>
<td>-monitor structures created upon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>synchronization</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CACAO</strong></td>
<td>-efficient</td>
<td>-mutex cache lookup</td>
<td>-mutex cache is contention point</td>
</tr>
<tr>
<td></td>
<td>- size : prop. to the number of parallel</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mutexes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thin-</strong></td>
<td>-efficient</td>
<td>-efficient(no lookup)</td>
<td>-scalable (decentralized)</td>
</tr>
<tr>
<td><strong>locks</strong></td>
<td>24 bits /object regardless if</td>
<td>-problems with busy waiting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>synchronization is used</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Meta-</strong></td>
<td>-efficient</td>
<td>-efficient(no lookup)</td>
<td>-scalable (decentralized)</td>
</tr>
<tr>
<td><strong>locks</strong></td>
<td>2 bits/object if synchronization is used</td>
<td>-no busy waiting</td>
<td></td>
</tr>
</tbody>
</table>
Complementary approach

• Static analysis for removing unnecessary lock operations
  – One monitor entered several times by the same thread
  – Enclosed monitors (one thread acquires the second monitor)
  – Monitor accessible only to one thread (eliminate lock operations)
  – Problems with dynamic class loading and reflection

• Papers
  – Aldrich, Chambers and comp., Static analyses for eliminating unnecessary synchronization from Java programs, SAS ‘99
  – Bogda, Hoelzle, Removing unnecessary synchronization in Java, OOPSLA ‘99