IBM Software Group

IBM

e business software

Java: Is it viable for High Performance Computing?

Bob Blainey

March 26, 2002

Strengths of Java

- Support for platforms ranging from servers to embedded devices
- Object-oriented
- Portable: write once, run everywhere
- Language support for multithreaded programming
- Automatic memory management (garbage collection)
- Dynamic binding
- Language support for error checking and structured exception handling
- Large set of standard libraries
- Majority of introductory programming classes are now taught in Java

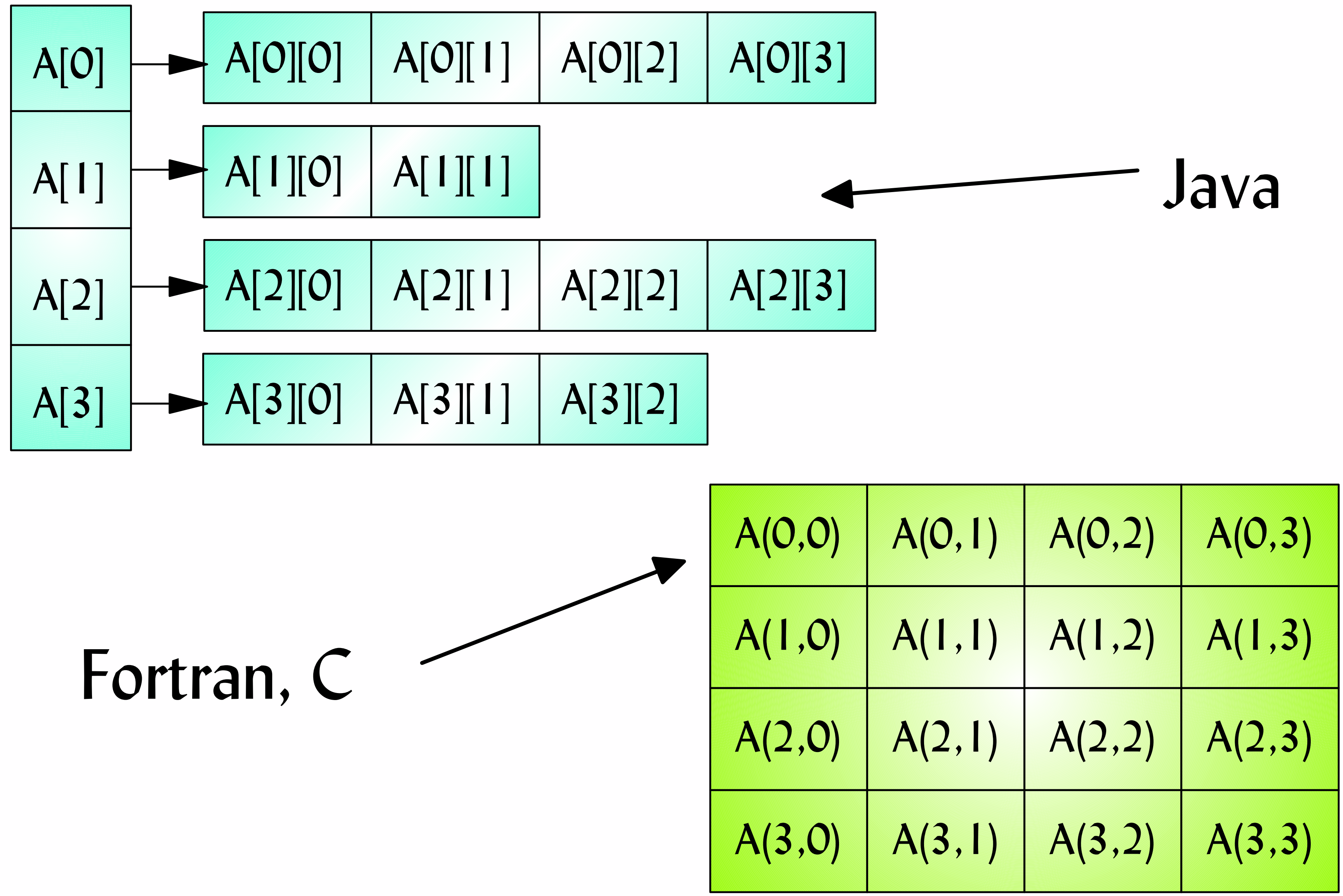
Challenges in Java Performance

- Inherited from OO programming: virtual method dispatch, space overhead of objects, interfaces
- Run-time checks (null pointer, array index out of bounds, dynamic typing)
- Impact of precise exceptions on optimization and path length
- Garbage collection overhead and memory usage
- Synchronization costs
- Lack of true multidimensional arrays (numerical computing)
- Bitwise reproducibility of results (floating point)
- Run-time binding (affects everything above!)

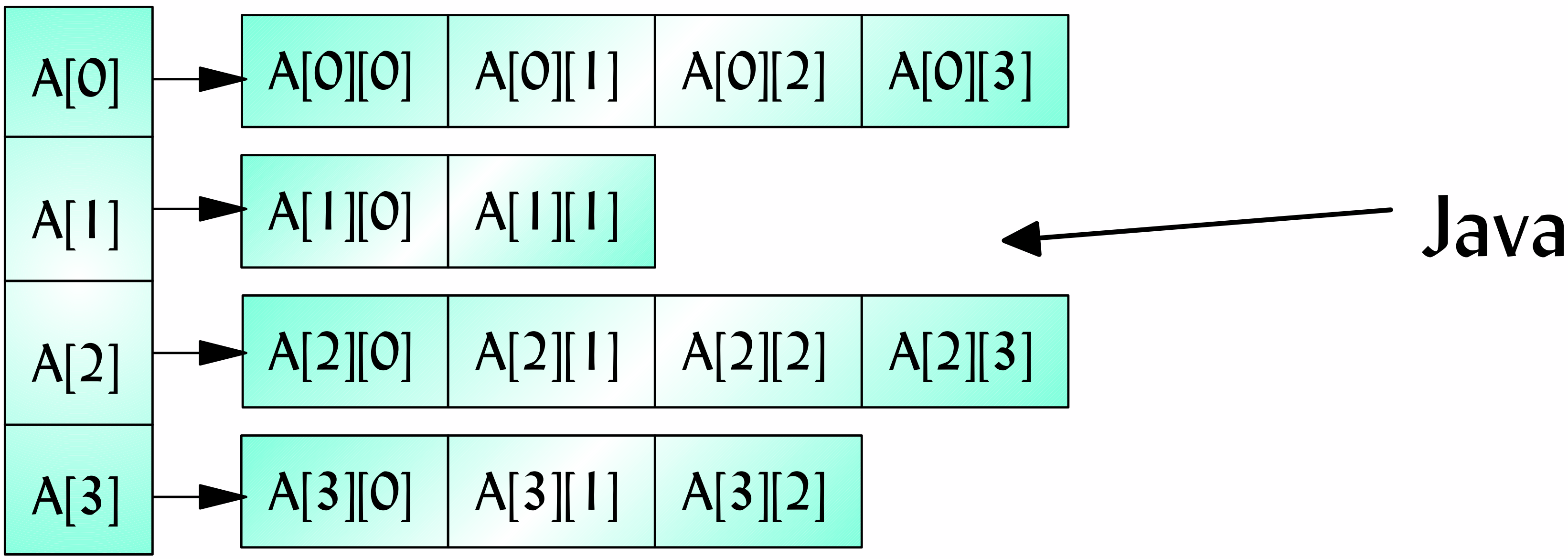
Java: More Missing Pieces for HPC

- Complex arithmetic
  - Need integrated support to avoid excessive object creation
  - Operator overloading to allow transparent complex/double computations
- Floating point standard
  - Too strict, need to allow fast floating point with acceptable if not bitwise identical results
  - e.g. fused multiply-add instruction not permitted
- Parallel programming
  - No support for parallel regions, loops, barriers, etc.
  - No support for SPMD programming model
- Faster Java Native Interface
  - Needed to utilize vast collection of legacy code

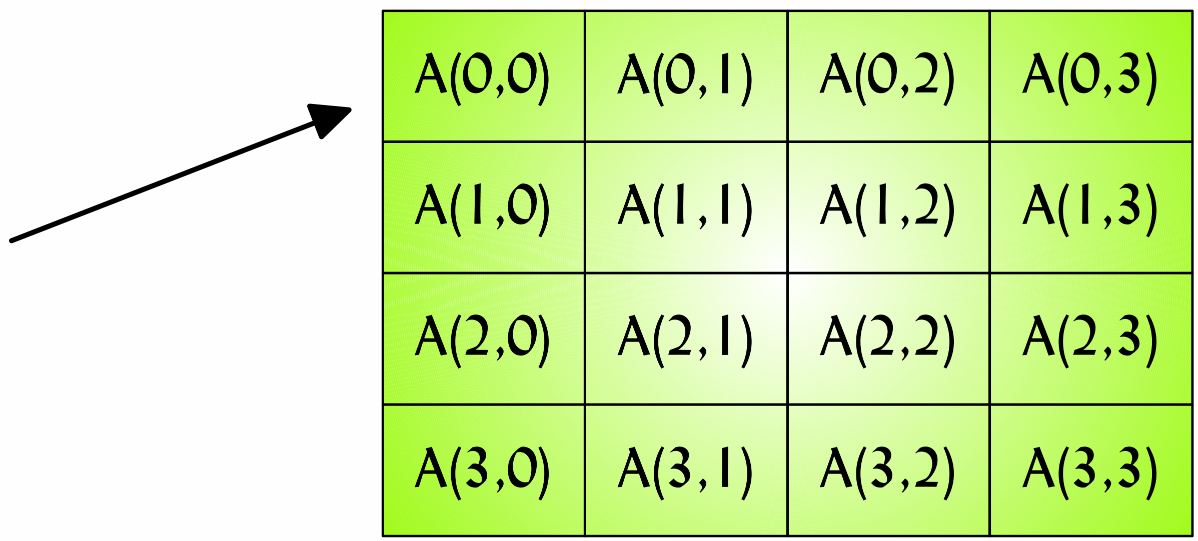
Array Layout



Array Layout



Fortran, C



Java Array Package

Array Package for Java

- Combining Fortran90 performance and functionality with Java safety and flexibility of array layout.
  - using dense storage internally.
  - operations on arrays and array sections.
  - no JVM change needed, 100% pure Java.
  - extensive checks for various exceptions (out of bounds, non-conforming arrays, invalid array shape).
  - internal array layout not exposed - more efficient representations may be used.
- A class for each elemental data type and rank, e.g. doubleArray2D, complexArray3D, intArray1D.
- More info:
  - http://www.jcp.org/jsr/detail/83.jsp
  - http://www.alphaworks.ibm.com/tech/ninja

Ref: Moreira, Midkiff, Gupta, Artigas, Snir, Lawrence. High performance numerical computing in Java. IBM Systems Journal, March 2000.

Exception Checks: What's the Problem?

```
n1:    a = b.f;     //  null pointer exception check.
n2:    x = y * z;
n3:    c = 1/x;     //  arithmetic exception (divide by zero) check.
```

Exception Checks: What's the Problem?

```
        try {
n1:         a = b.f;     //  null pointer exception check.
n2:         x = y * z;
n3:         c = 1/x;     //  arithmetic exception (divide by zero) check.
        } catch (NullPointerException e) {
          System.out.println(x);
          e.printStackTrace();
        } catch (ArithmeticException e) {
          ....
```

Exception Checks: What's the Problem?

```
        try {
n1:         a = b.f;     //  null pointer exception check.
n2:         x = y * z;
n3:         c = 1/x;     //  arithmetic exception (divide by zero) check.
        } catch (NullPointerException e) {
          System.out.println(x);
          e.printStackTrace();
        } catch (ArithmeticException e) {
          ....
```

Cannot move n1 after n2 or n3, in spite of no data dependence.

Exception Checks: What's the Problem?

```
      try {
n1:       a = b.f;      //  null pointer exception check.
n2:       x = y * z;
n3:       c = 1/x;      //  arithmetic exception (divide by zero) check.
      } catch (NullPointerException e) {
        System.out.println(x);
        e.printStackTrace();
      } catch (ArithmeticException e) {
        ....
```

Exceptions are precise in Java.

- nice language feature - robustness, portability.
- bad for performance.

Why Should You Care?

- Potentially excepting instructions (PEIs) are very common in Java programs.
  - e.g., read/write of fields of objects, arrays loads and stores, method calls, object allocations, type casts.

- Precise exceptions introduce many false dependences.
  - to ensure program state at exception point is "correct".
  - to ensure the "correct" exception is thrown.

- This hampers optimizations that reorder instructions, like instruction scheduling, instruction selection across PEI, loop transformations, parallelization.
  Can lead to bad performance.

Basic Intuition

- Program state that needs to be preserved (for correct execution) when exception is thrown is often quite small.
  - print an error message and exit.
  - throw away results from exception throwing computation and fall back to some default approach.

- Runtime exceptions should be thrown rarely.
  - optimize program for the case when exception is not thrown.

Overcoming Exception Sequence Dependences

- Generate two sets of code.
- Optimized code:
  - completely ignores exception sequence dependences.
  - may throw an "incorrect" exception.
- Compensation code:
  - executes only if optimized code throws an exception.
  - intercepts the exception, and throws the correct exception.
  - does not require any check-pointing in the optimized code to recover the correct exception - very low overhead in the expected case.

Array Access: Exception Checks

- Consider standard dot-product matrix-multiply:

```
for (int i=0; i<m; i++)
  for (int j=0; j<p; j++)
    for (int k=0; k<n;k++)
      C[i][j] += A[i][k]*B[k][j];
```

- Each iteration requires 6 null-pointer checks (**C**, **C[i]**, **A**, **A[i]**, **B**, **B[k]**) and 6 index checks (**i** and **j** for **C**, **i** and **k** for **A**, **k** and **j** for **B**).
- The possibility of exceptions prevents any iteration reordering.

Safe Region Creation

```
if ((C != null) && (A != null) && (B != null) &&
     (m-1 < C.size(0)) && (n-1 < C.size(1)) &&
     (m-1 < A.size(0)) && (p-1 < A.size(1)) &&
     (p-1 < B.size(0)) && (n-1 < B.size(1))) {

   for (i=0; i<m; i++)
      for (j=0; j<n; j++)
         for (k=0; k<p; k++)
            C[i,j] = C[i,j] + A[i,k] * B[k,j] ;
} else {

   for (i=0; i<m; i++)
      for (j=0; j<n; j++)
         for (k=0; k<p; k++)
            C[i,j] = C[i,j] + A[i,k] * B[k,j];
```

versioning test

safe region: no exception checks

unsafe region: with exception checks

Need for Alias Disambiguation

```
if ((C != null) && (A != null) && (B != null) &&
     (m-1 < C.size(0)) && (n-1 < C.size(1)) &&
     (m-1 < A.size(0)) && (p-1 < A.size(1)) &&
     (p-1 < B.size(0)) && (n-1 < B.size(1))) {

   for (i=0; i<m; i++)
      for (j=0; j<n; j++)
         for (k=0; k<p; k++)
            C[i, j] = C[i, j] + A[i, k] * B[k, j] ;
}
```

versioning test

safe region: no exception checks

Can apply loop transformations for locality enhancement or parallelization in safe region only if array C is not aliased with A or B.

Key Property of Java

Pointers - object references only:

```
p = new Object();
p = new int[100];
```

Cannot have statements like:

```
q = & x;
q = & p[i];
```

Therefore, two variables (objects) dereferenced via Java pointers cannot overlap partially: must be either identical or non-overlapping.

Two Java 1D arrays / Array package objects cannot overlap partially.

Alias Disambiguation via Versioning

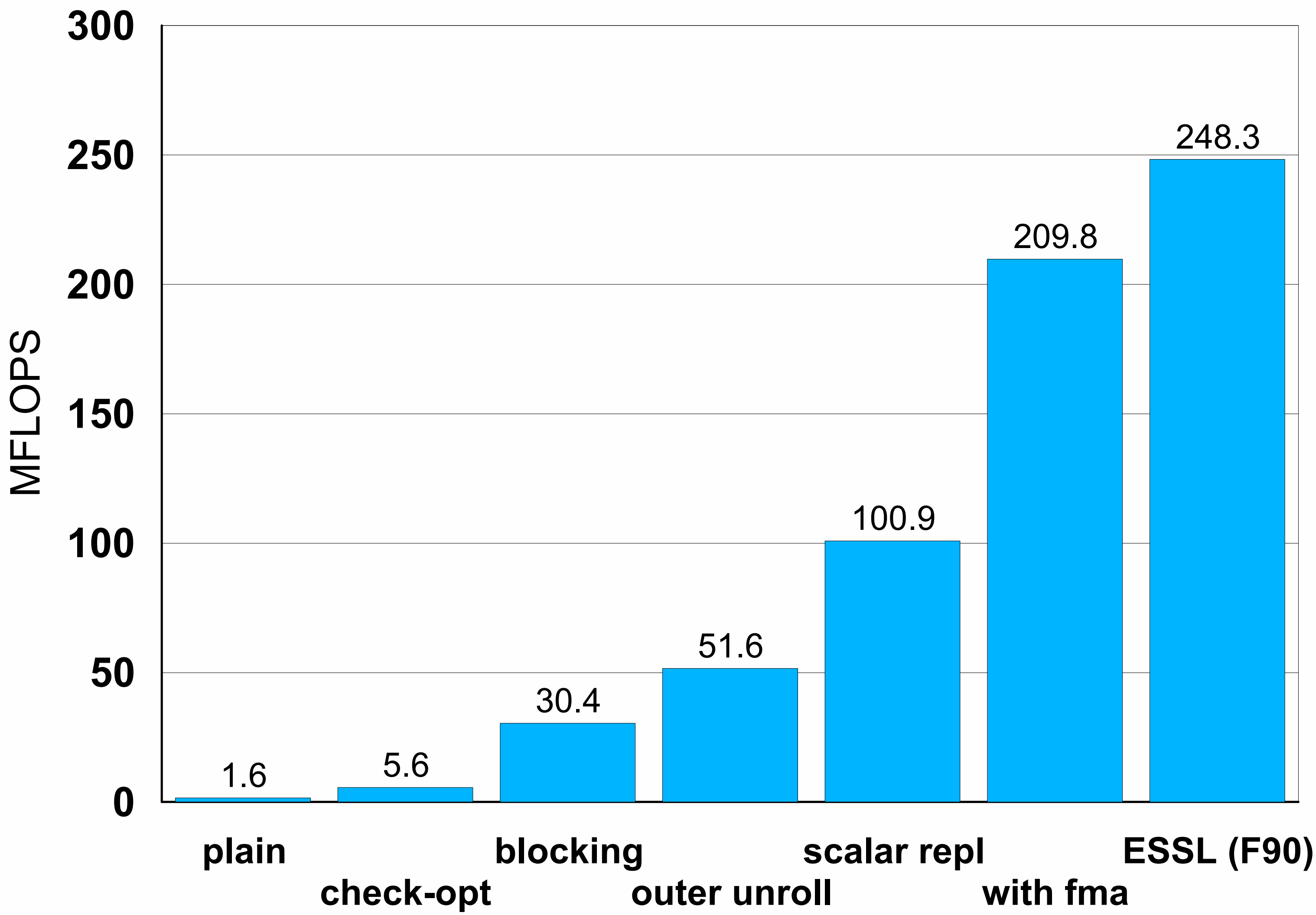
```
if ((C != null) && (A != null) && (B != null) &&
    (m-1 < C.size(0)) && (n-1 < C.size(1)) &&
    (m-1 < A.size(0)) && (p-1 < A.size(1)) &&
    (p-1 < B.size(0)) && (n-1 < B.size(1))) {

  if (C.data != A.data  &&  C.data != B.data) {
    for (i=0; i<m; i++)
      for (j=0; j<n; j++)
        for (k=0; k<p; k++)
          C'[i, j] = C'[i, j] + A'[i, k] * B'[k, j] ;
  } else {
    for (i=0; i<m; i++)
      for (j=0; j<n; j++)
        for (k=0; k<p; k++)
          C[i, j] = C[i, j] + A[i, k] * B[k, j] ;
  }
}
```

safe, alias-free region: can apply loop transformations

introduce new symbols with more precise alias information.

500x500 MATMUL on RS/6000 590



Ref: Moreira, Midkiff, Gupta. From flop to megaflops: Java for technical computing. ACM TOPLAS 2000.

Complex numbers in Java

- Java has no complex primitive data type.
  - Solution: standard Complex class (Java Grande).
- Treating complex numbers as objects results in too much overhead.
- Example: dot product

```
Complex[] a,b; Complex s;
for (i=0; i<n; i++)
  s.assign(s.plus(a[i].times(b[i])));
```

generates 2n temporary Complex objects!

Semantic Expansion of Complex Class

- Complex class declared final.
- Most methods (like plus, minus, times) expanded to operate directly on complex values rather than objects.
- Complex value converted lazily into object if object-oriented operation (not semantically expanded) performed on it.
- Synergy with semantic expansion of Array package: get benefits of true multidimensional arrays of complex values.

Ref: Wu, Midkiff, Moreira, Gupta. Efficient handling of complex numbers in Java. ACM Java Grande 1999.

Escape Analysis

- Generalizing the idea of optimizing object creation and management
- Focus on objects that do not *escape* a given scope such as method or thread of creation. An object escapes if there may be some reference to it outside the scope.
- A method-local object can be allocated on the method stack:
  - inherently more efficient than heap allocatation
  - storage automatically reclaimed when method exits
  - in many cases, method-local objects can be allocated to machine registers
- A thread-local object need not be locked for mutual exclusion in synchronized method/statement.

Ref: Choi, Gupta, Serrano, Sreedhar, Midkiff. Escape analysis for Java. OOPSLA 1999.

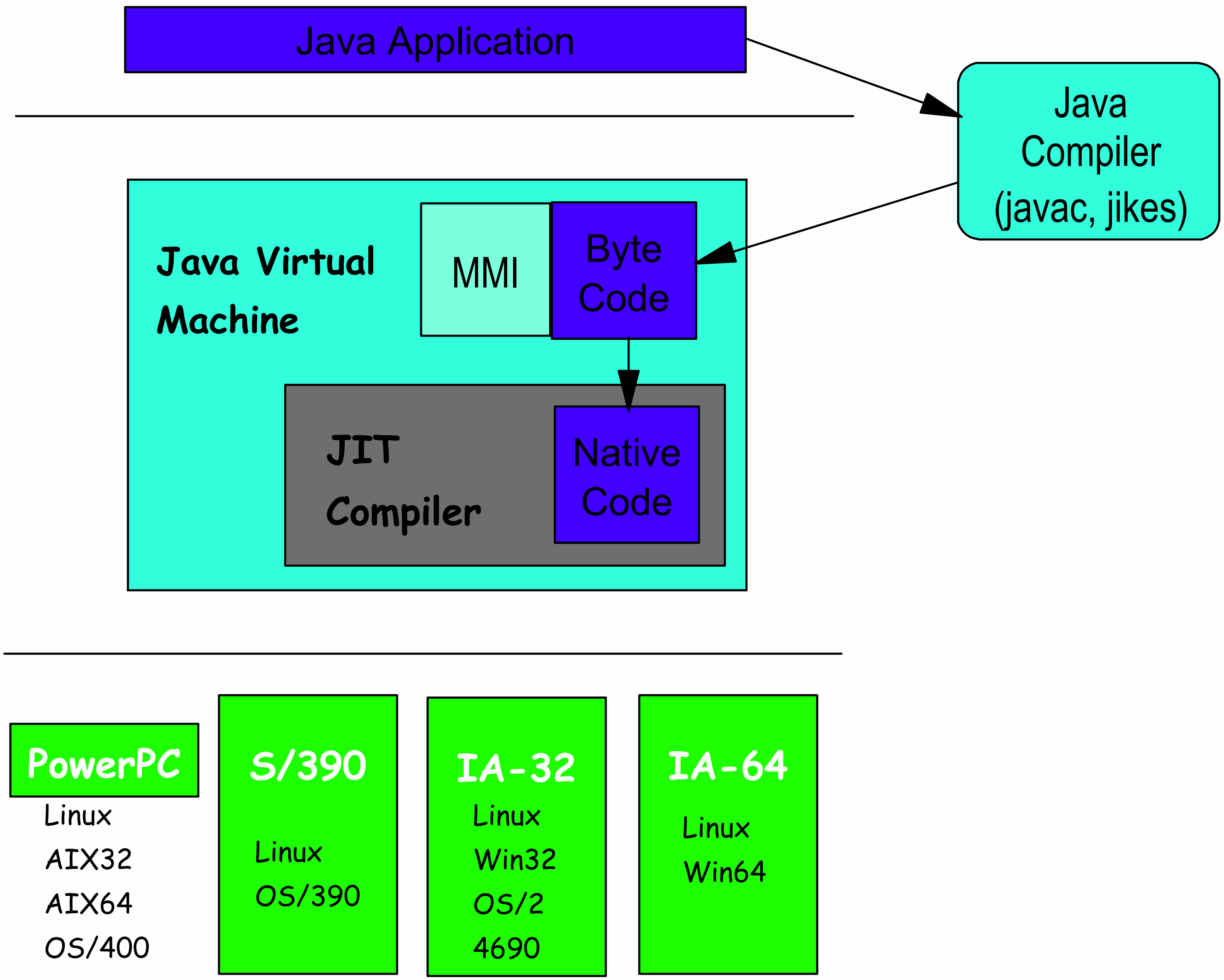
Object Inlining

- Going a step further in escape analysis leads us to the idea of object inlining
- If an object is reachable exclusively via some other unique object, then the objects can be coalesced into a single object.
  - Leads to more efficient space utilization
  - Pervasive application of object inlining leads to a systematic reduction in memory management (garbage collection) overhead
- Requires escape analysis to determine whether an object reference is reachable from another object or from some local reference

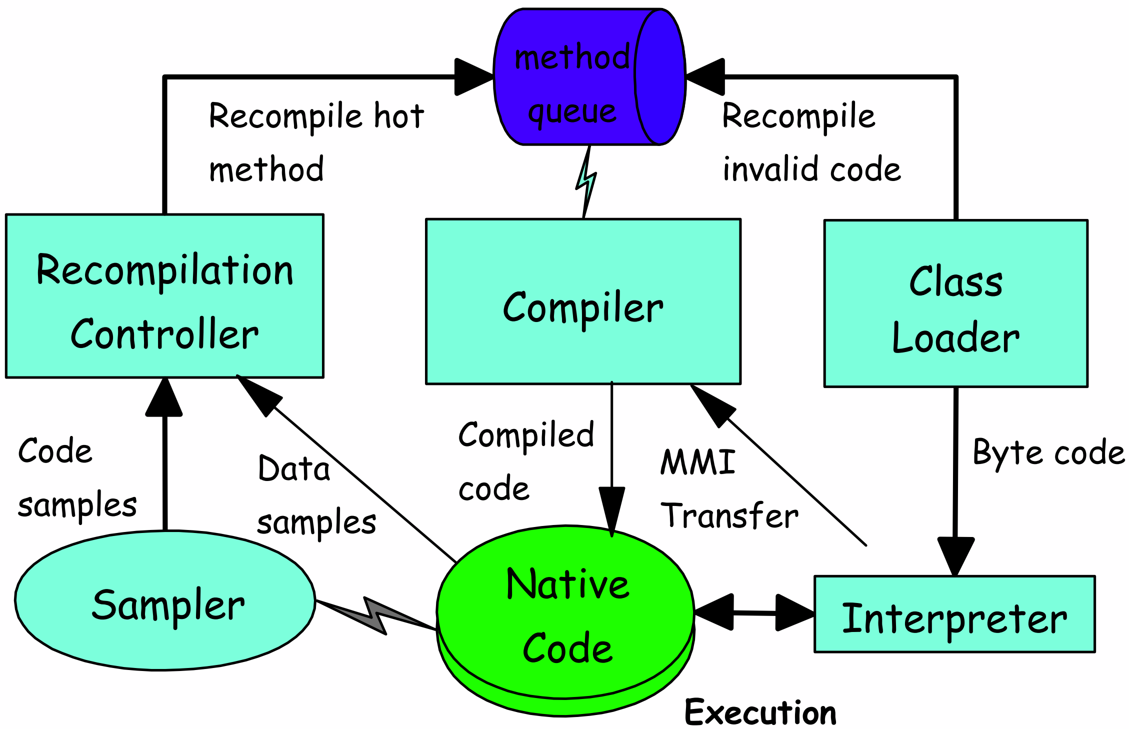
Some Performance Coding Practices

- Replace field and array references with locals where possible
  - Minimizes null pointer and array bound checks explictly
- Avoid synchronization where possible
  - Language facilities make it easy to apply too much synchronization
  - Consider rewriting synchronized methods as a synchronized wrapper (callable from outside the object) and an unsynchronized body (callable from other synchronized methods on this object)
  - "Coarsen" locks where possible by piggybacking on other object locks or combining adjacent synchronized code
- Try to avoid false sharing
  - eg. PowerPC reservation granule is 128 bytes - don't pack shared data any closer than that
- Use the largest heap you can to minimize garbage collection effects
  - Can increase average pause time (and therefore response time)
  - Heap size can determine garbage collection algorithm used (eg. generational vs. mark & sweep)
- Scope references to objects as tightly as possible
  - Object space may be recycled more quickly

IBM JDK Architecture



IBM JIT Compilation Cycle



Interpreter

Method invocation counts
Conditional path info
Loop detection

→

Fast startup
Class & method resolution
Class initialization

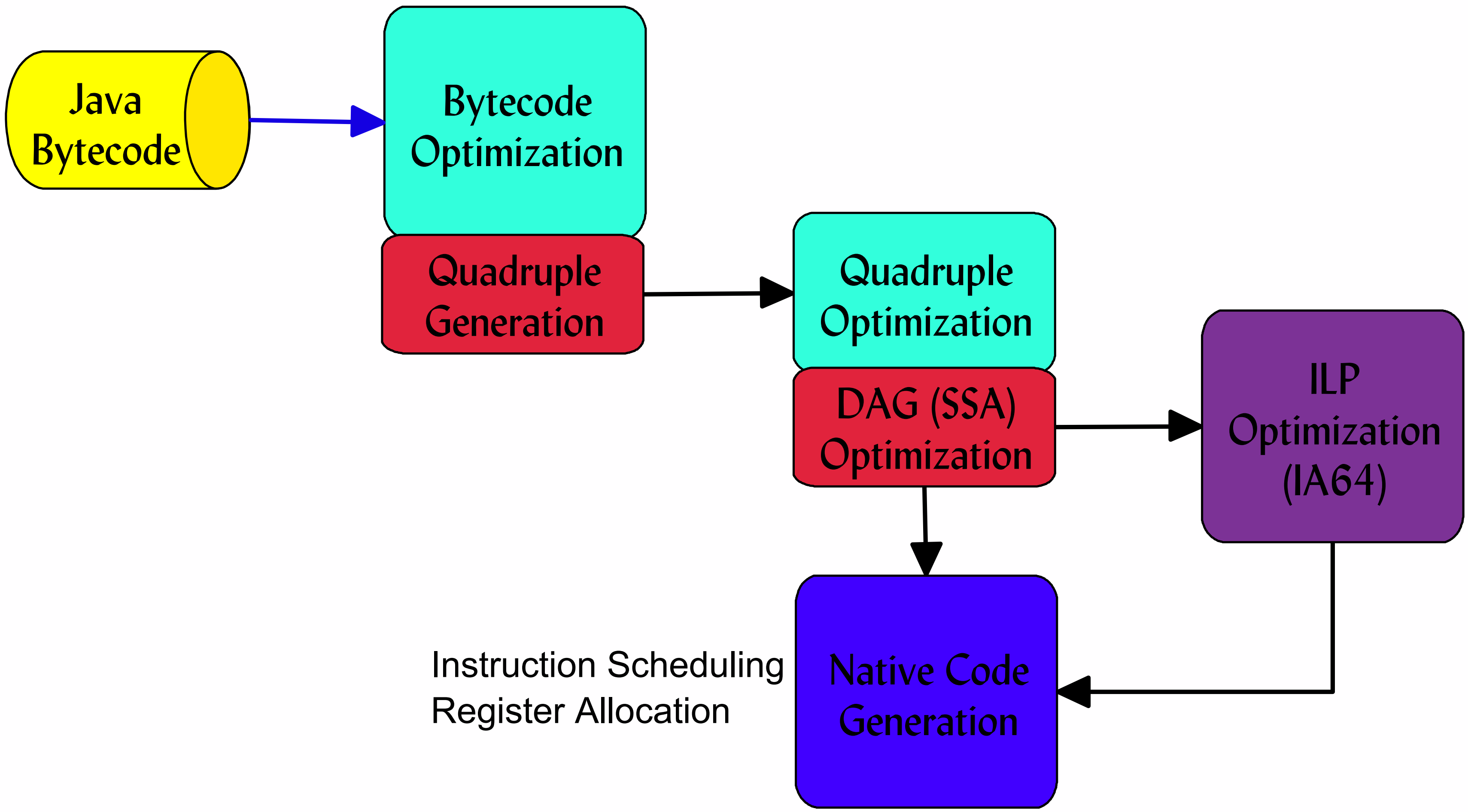
Sampler/Compiler

Hot methods
Common parameters

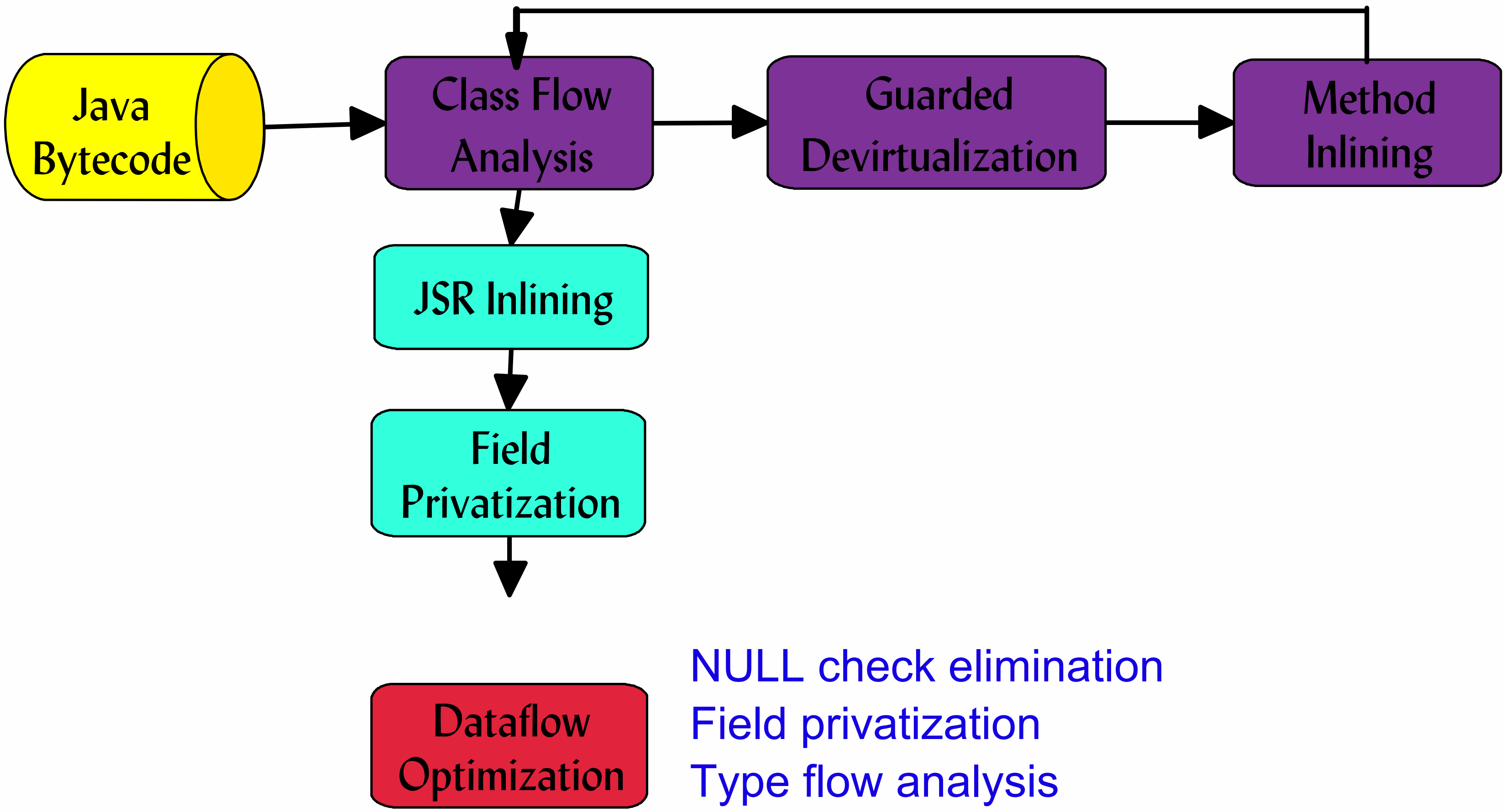
→

Good code for warm methods
Best code for hot methods
Specialized hot methods

Inside the IBM JIT

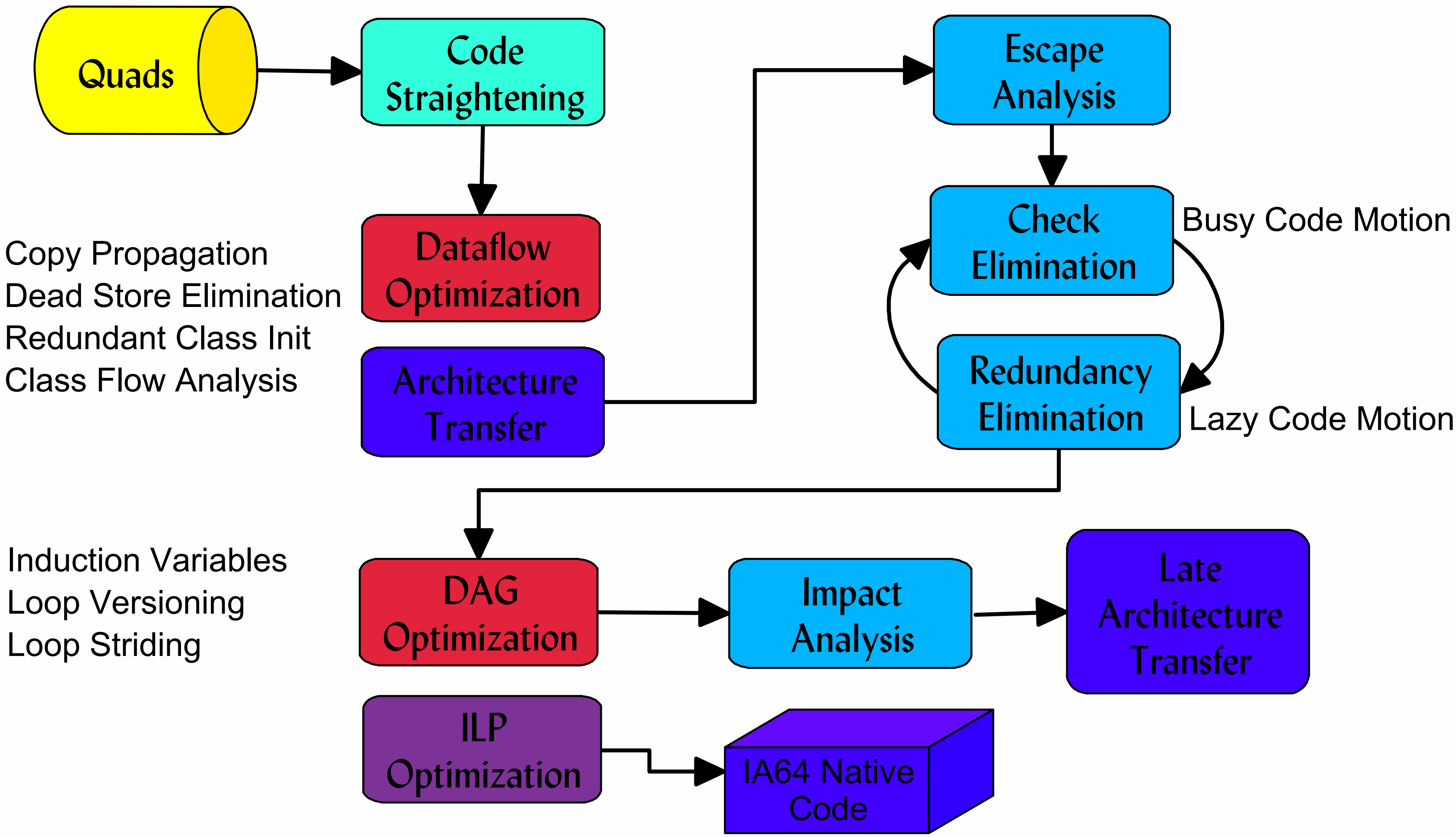


Bytecode Optimization

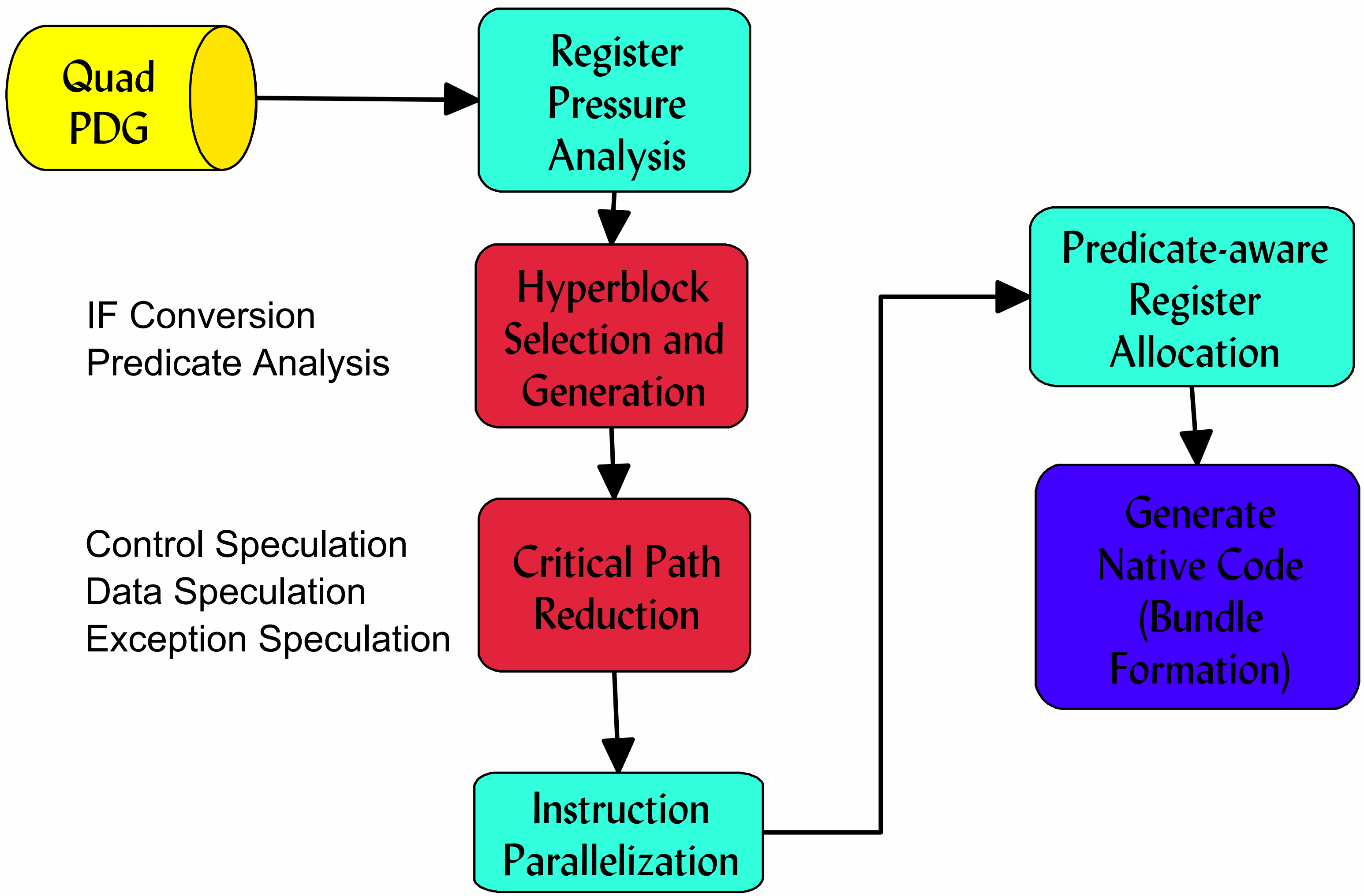


NULL check elimination
Field privatization
Type flow analysis
Array check elimination

Quadruple Optimization



Instruction-Level Parallel Optimization (IA-64)



Questions and Answers