

Optimizing Java Programs: Challenges and Opportunities

Manish Gupta

IBM T. J. Watson Research Center
Yorktown Heights, NY

<http://www.research.ibm.com/ninja>

<http://www.research.ibm.com/quicksilver>

<http://www.research.ibm.com/jalapeno>

Joint Work with:

- Pedro Artigas
- Rajesh Bordawekar
- Jong-Deok Choi
- Michael Hind
- Samuel Midkiff
- Jose Moreira
- Mauricio Serrano
- Marc Snir
- V. C. Sreedhar
- Peng Wu

Motivation: Java is Gaining Popularity

Platforms ranging from servers to embedded devices.

- ▶ Object-oriented
- ▶ Portable - write once, run everywhere
- ▶ Multithreaded
- ▶ Garbage collected
- ▶ Dynamically loaded
- ▶ Exception-handling
- ▶ Large set of standard libraries
- ▶ Majority of introductory programming classes are now taught in Java

Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, space overhead of objects.
- Run-time checks (null pointer, array index out of bounds,...)
- Precise exceptions.
- Automatic memory management: garbage collection.
- Synchronization costs.
- Dynamic class loading.
- Memory model too strong.
- Lack of true multidimensional arrays (numerical computing).
- Run-time binding (affects everything above!).

Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, space overhead of objects.
- **Run-time checks (null pointer, array index out of bounds,...)**
- **Precise exceptions.**
- Automatic memory management: garbage collection.
- Synchronization costs.
- Dynamic class loading.
- Memory model too strong.
- Lack of true multidimensional arrays (numerical computing).
- Run-time binding (affects everything above!).

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.

Why?

```
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.
- Precise exceptions introduce many 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.

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.

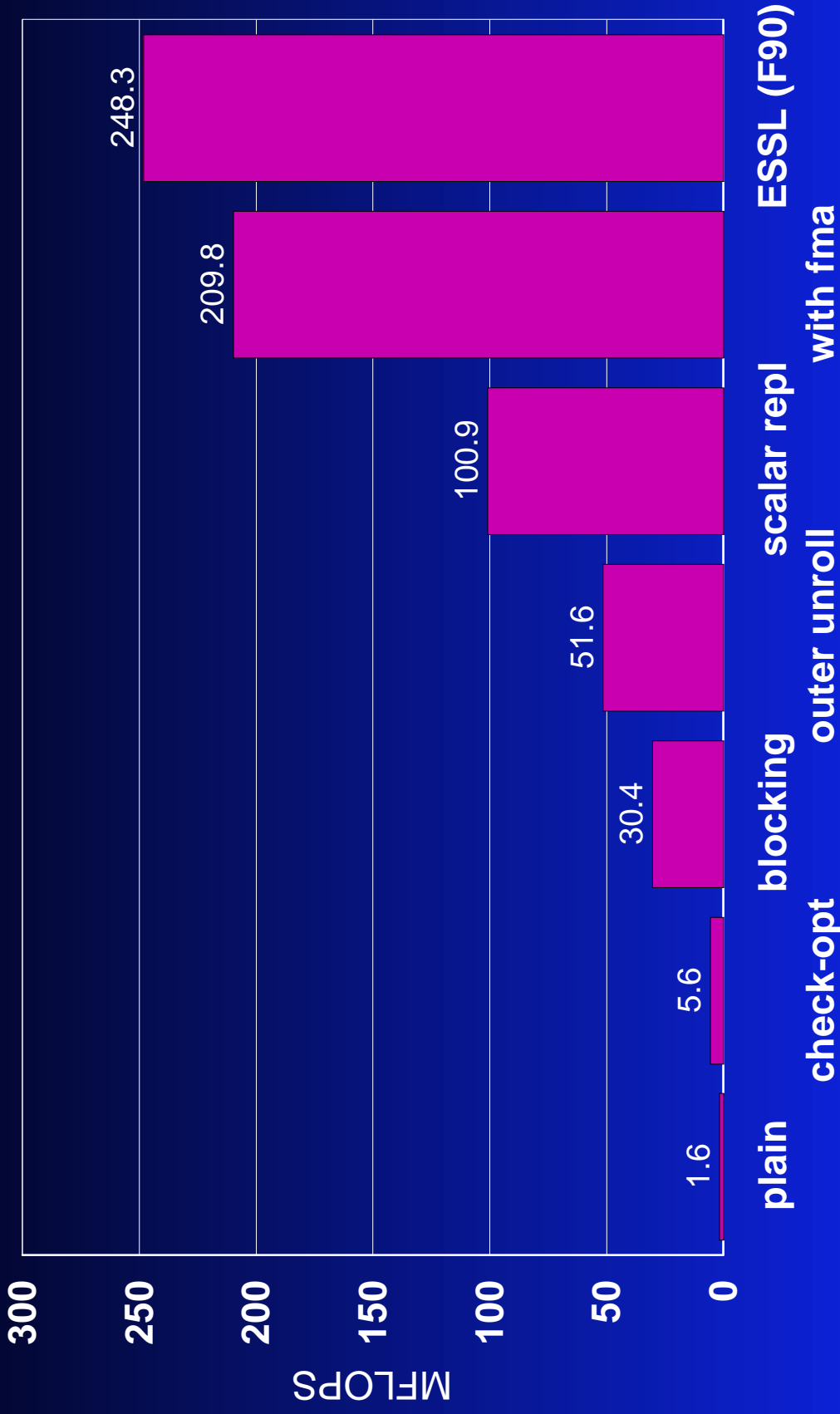
Array Access: Exception Checks

- 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.
- On RS/6000 590: Java 1.6 Mflops, Fortran 220 Mflops.

500x500 MATMUL on RS/6000 590



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

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.

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.

Program State Dependences

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

Elimination of Redundant Program State Dependences

- Liveness analysis at exception handler.
- Interprocedural propagation of liveness information.
 - ▶ dynamic analysis - more accurate, runtime overhead.
 - ▶ static analysis - less accurate, no runtime overhead.
 - ▶ either of above analyses may be used in a static/dynamic compiler.

Ref: Gupta, Choi, Hind. Optimizing Java programs in the presence of precise exceptions, ECOOP 2000.

Dynamic Analysis

- Propagate liveness information about enclosing exception handlers (LEEH) to each method activation.
 - ▶ Extra LEEH parameter - single word.
- Identify set of exceptions for which info is encoded:
 - ▶ possibly thrown by program - linear scan of program, or
 - ▶ predefined set of exceptions - RuntimeException; catchall for others.
- At a call site inside *try* block:
 - ▶ $LEEH_{\text{callee}} = (LEEH_{\text{caller}} \& \text{KILL}) \mid \text{GEN}$
 - ▶ GEN is encoding of liveness info for different exception types at *catch(-finally)* blocks.
 - ▶ KILL masks out info for exceptions definitely caught outside the call.
- At a call site not inside *try* block:
 - ▶ $LEEH_{\text{callee}} = LEEH_{\text{caller}}$

Dynamic Analysis: Percentage of Method Calls with No/Trivial Enclosing Exception Handlers



Exception Sequence Dependences

```
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) {  
    ....  
}
```

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

- Specialized solution for Array accesses

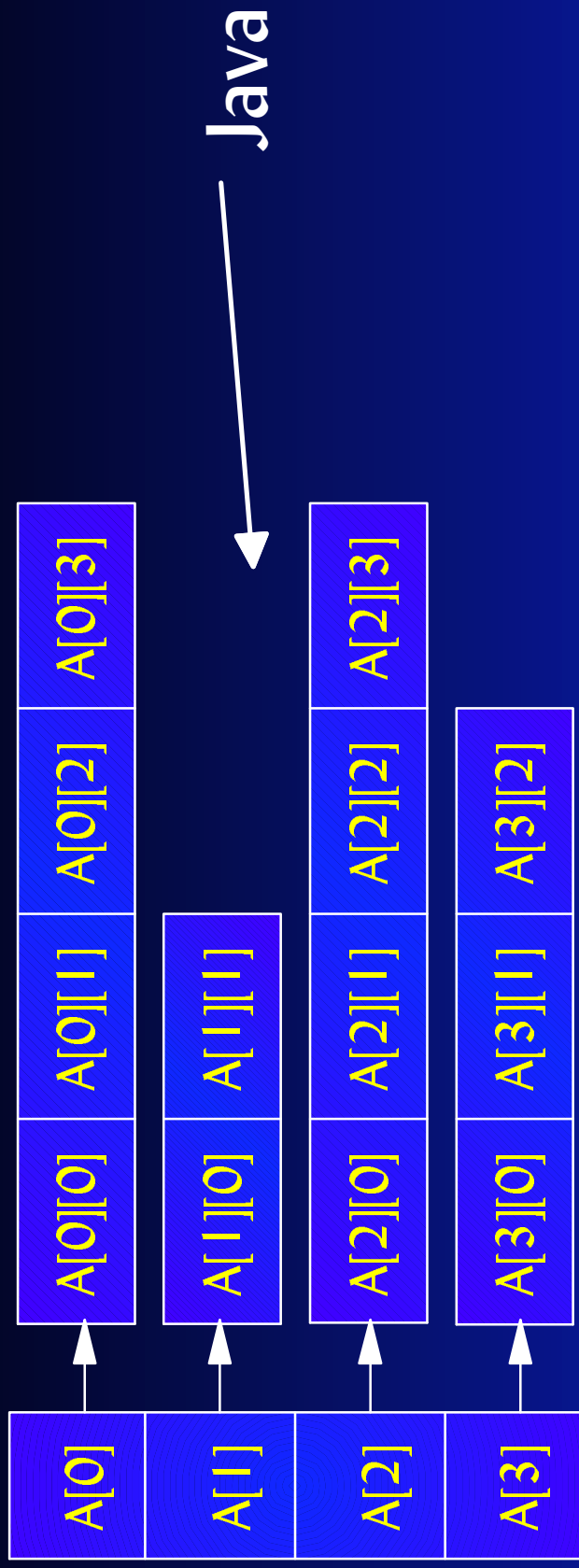
```
for (int i=0; i<m; i++)  
  for (int j=0; j<p; j++)  
    for (int k=0; k<n;k++)  
      C[i][j] += A[i][k]*B[k][j];
```

- On RS/6000 590: Java 1.6 Mflops, Fortran 220 Mflops.
- 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.

Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, space overhead of objects.
- **Run-time checks (null pointer, array index out of bounds,...)**
- **Precise exceptions.**
- Automatic memory management: garbage collection.
- Synchronization costs.
- Dynamic class loading.
- Memory model too strong.
- **Lack of true multidimensional arrays (numerical computing).**
- Run-time binding (affects everything above!).

Array Layouts

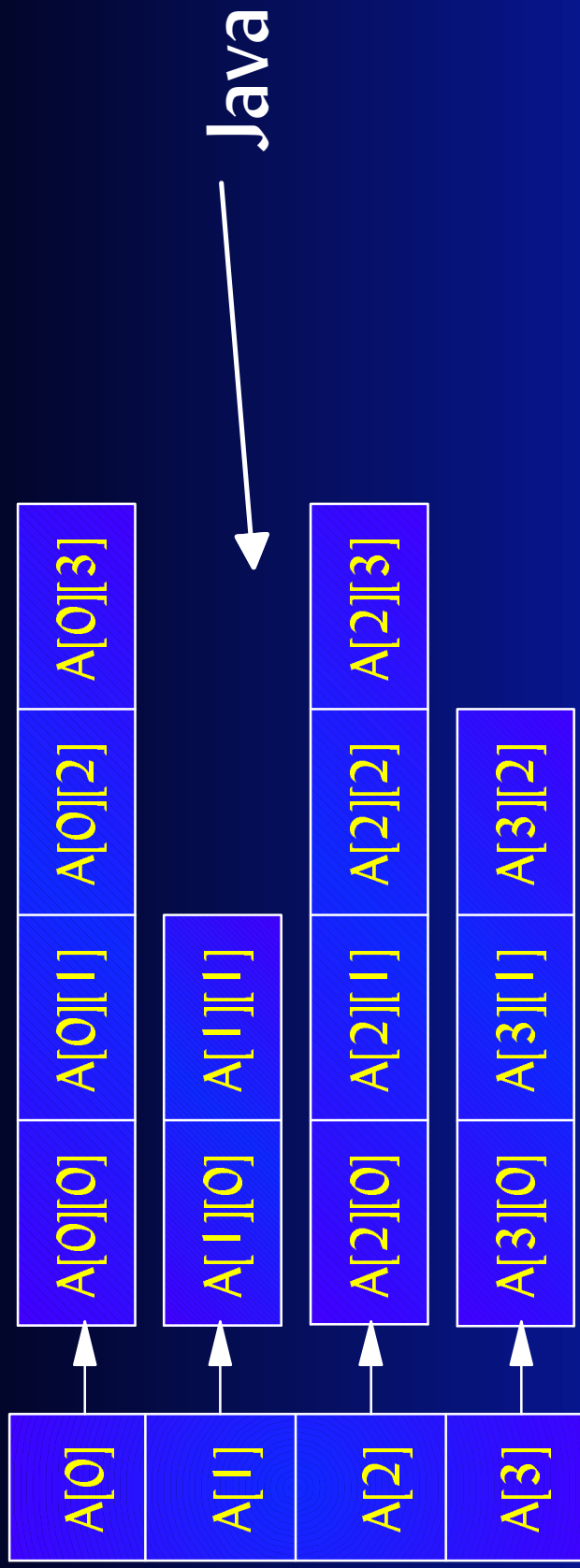


Fortran, C

$A(0,0)$	$A(0,1)$	$A(0,2)$	$A(0,3)$
$A(1,0)$	$A(1,1)$	$A(1,2)$	$A(1,3)$
$A(2,0)$	$A(2,1)$	$A(2,2)$	$A(2,3)$
$A(3,0)$	$A(3,1)$	$A(3,2)$	$A(3,3)$

The diagram illustrates the column-major layout of a 4x4 array. The text "Fortran, C" is written vertically on the left. A white arrow points from this text to the top-left corner of a 4x4 grid of elements. The elements in the grid are: $A(0,0)$, $A(0,1)$, $A(0,2)$, $A(0,3)$ in the first row; $A(1,0)$, $A(1,1)$, $A(1,2)$, $A(1,3)$ in the second row; $A(2,0)$, $A(2,1)$, $A(2,2)$, $A(2,3)$ in the third row; and $A(3,0)$, $A(3,1)$, $A(3,2)$, $A(3,3)$ in the fourth row.

Array Layouts



A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)
A(3,0)	A(3,1)	A(3,2)	A(3,3)

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.
- Available for free download from:
<http://www.alphaWorks.ibm.com>

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

Array Access Exception Checks

Matrix Multiplication Example

```
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] ;
```

Array Access Exception Checks

Matrix Multiplication Example

```
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];
```

- Null pointer checks and array index out of bounds checks.
- Precise exception requirement - compiler does not reorder instructions around potential exception points.

Ref: Artigas, Gupta, Midkiff, Moreira. High performance numerical computing: Language and compiler issues, LCPC 99.

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))) {
```

versioning test

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    for (k=0; k<p; k++)
```

safe region: no
exception checks

```
        C[i, j] = C[i, j] + A[i, k] * B[k, j];
    }
```

Can apply loop transformations for locality enhancement/
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];
```

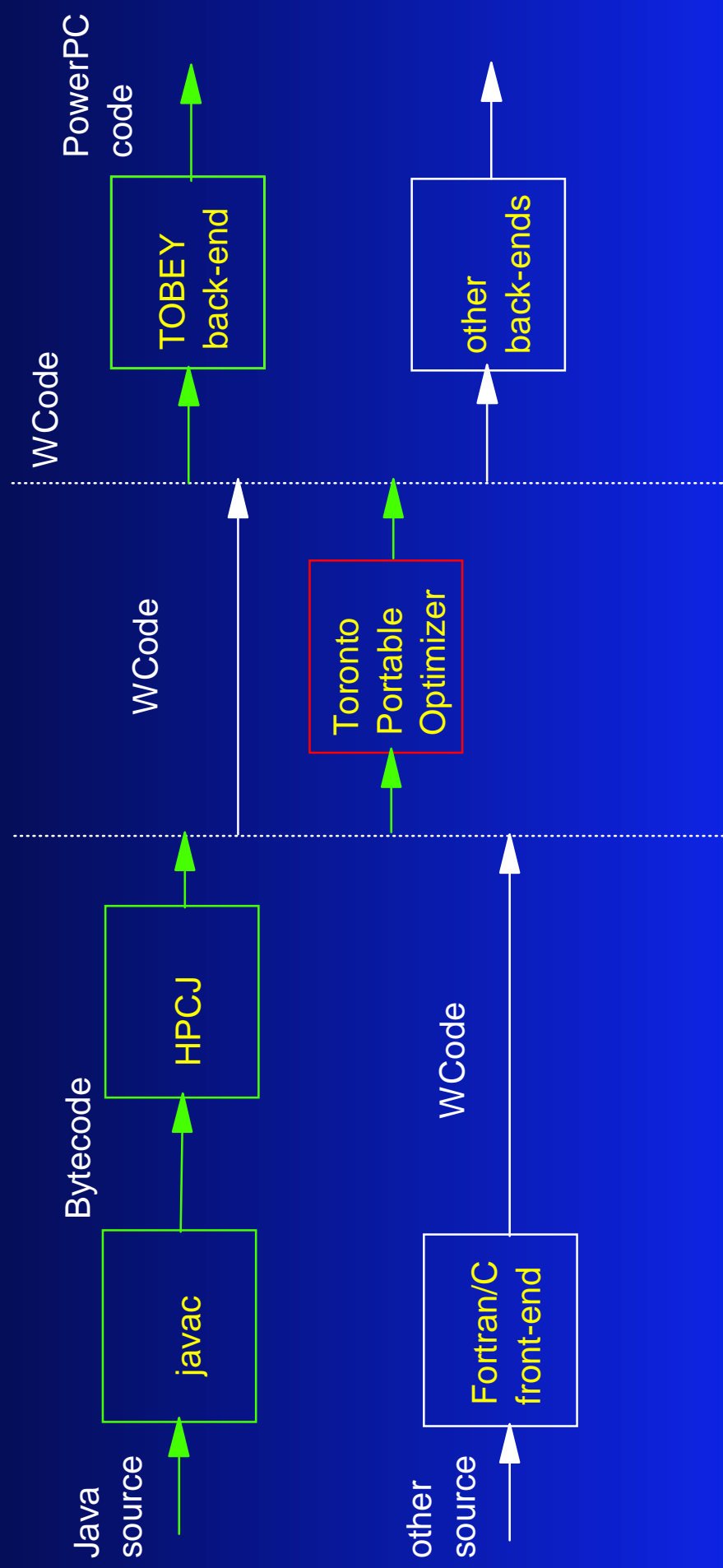
```
            }
```

```
    }
```

safe, alias-free region: can
apply loop transformations

introduce new symbols
with more precise alias
information.

Implementation in IBM Compiler



Ref: Artigas, Gupta, Mickiff, Moreira. Automatic loop transformations for Java. ICSE 2000

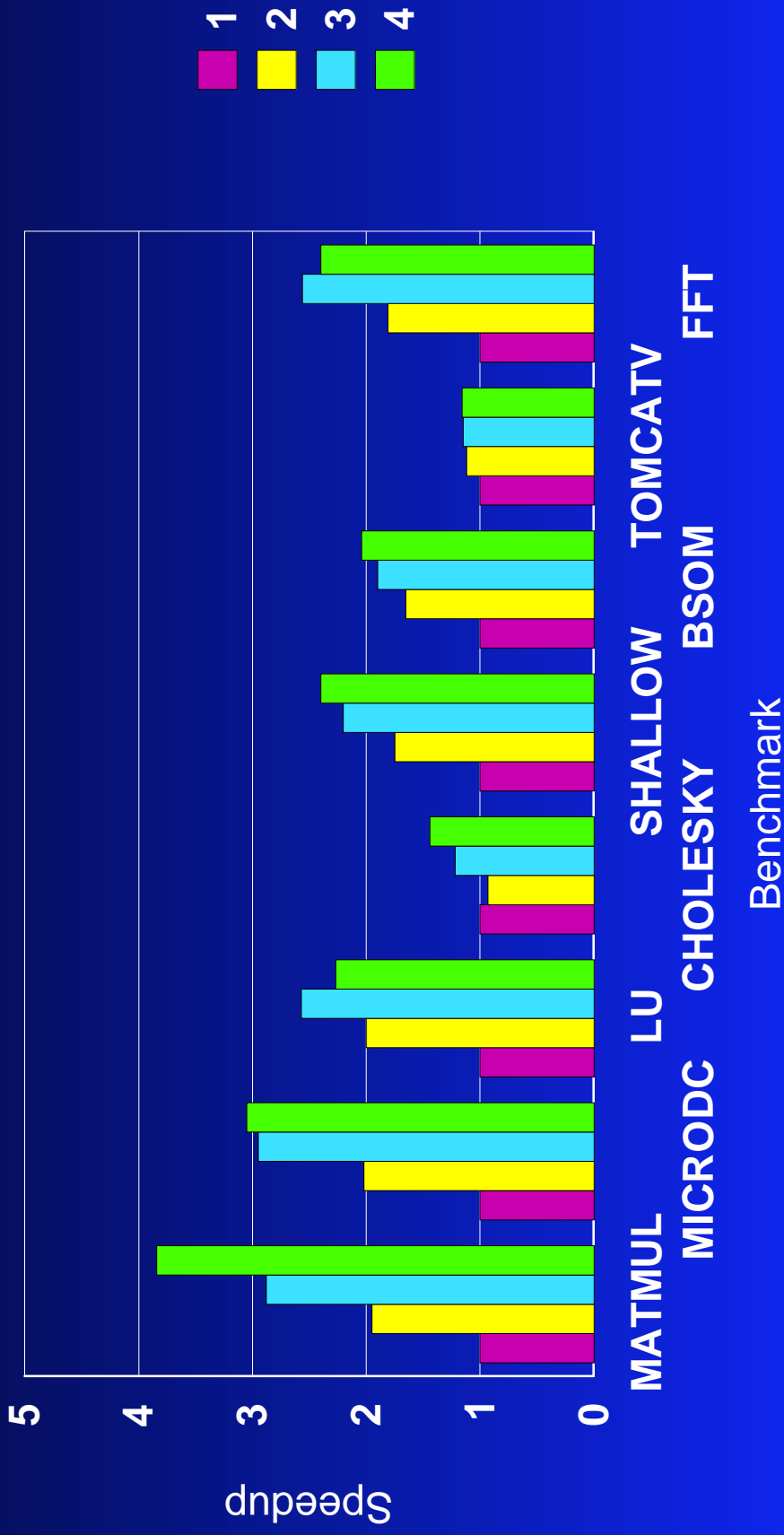
Serial Execution Performance

Performance on 200 MHz POWER 3



Automatic Parallelization Results

Speedups on 4-way POWER3 SMP



Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, **space overhead of objects**.
- Run-time checks (null pointer, array index out of bounds,...)
- Precise exceptions.
- Automatic memory management: garbage collection.
- Synchronization costs.
- Dynamic class loading.
- Memory model too strong.
- Lack of true multidimensional arrays (numerical computing).
- Run-time binding (affects everything above!).

Complex numbers in Java

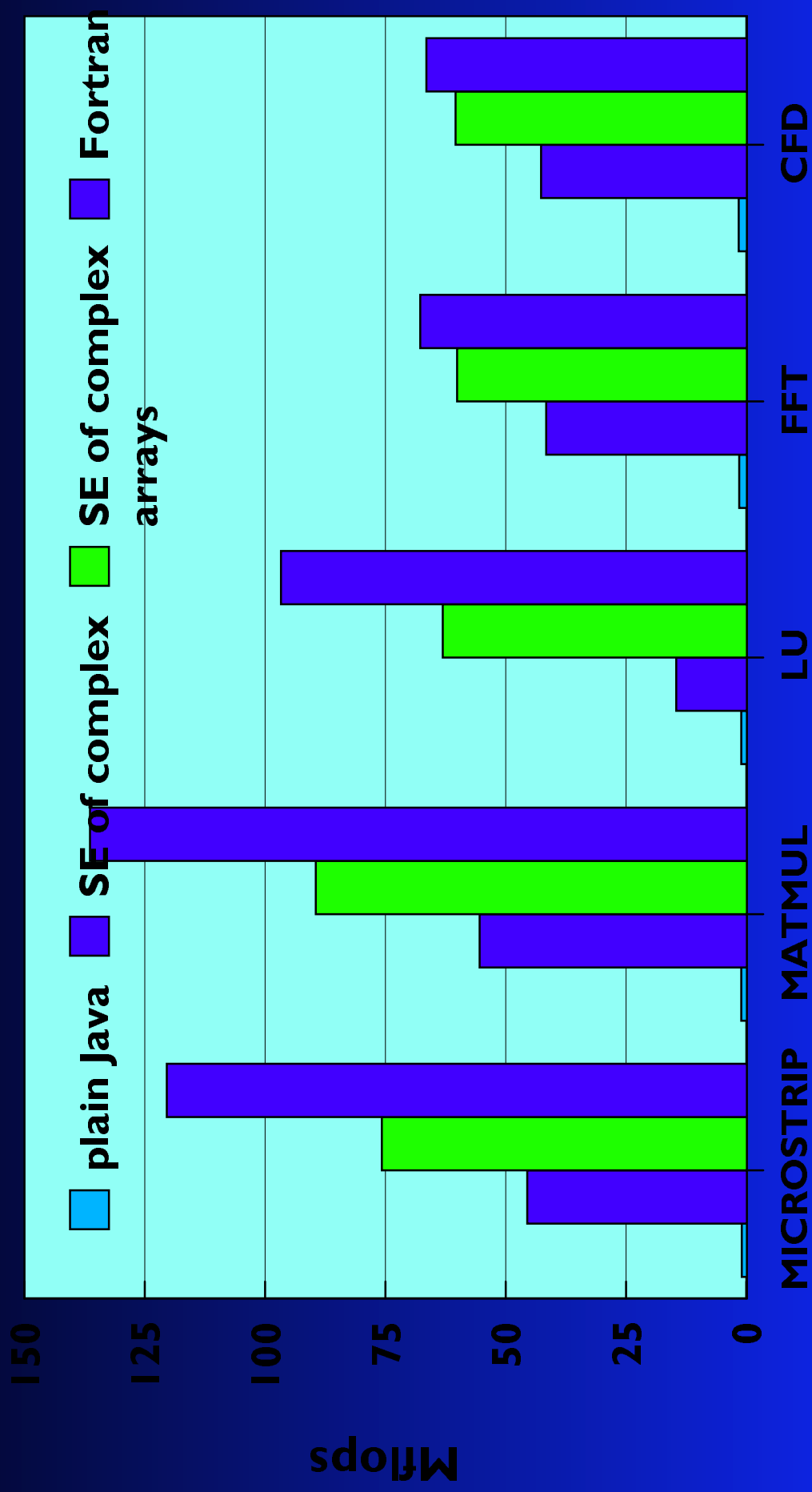
- Java has no complex primitive data type. Solution: standard Complex class (Java Grande).
- Treating complex numbers as objects results in too much overhead.
- Example: dot product
`Complex[] a,b; Complex s;`
`for (i=0; i<n; i++)`
`s.assign(a[i].times(b[i]));`
generates 2n temporary Complex objects!

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. Java Grande 1999.

Java semantic expansion on RS/6000 590



Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, space overhead of objects.
- Run-time checks (null pointer, array index out of bounds,...)
- Precise exceptions.
- **Automatic memory management: garbage collection.**
- **Synchronization costs.**
- Dynamic class loading.
- Memory model too strong.
- Lack of true multidimensional arrays (numerical computing).
- Run-time binding (affects everything above!).

What is Escape Analysis?

- Analysis to identify objects that do not escape a given scope such as **method** or **thread** of creation.
- A **method-local** object can be allocated on the method stack:
 - ▶ inherently more efficient than heap allocation.
 - ▶ storage automatically reclaimed when method exits.
- A **thread-local** object need not be locked for mutual exclusion in synchronized method/statement.

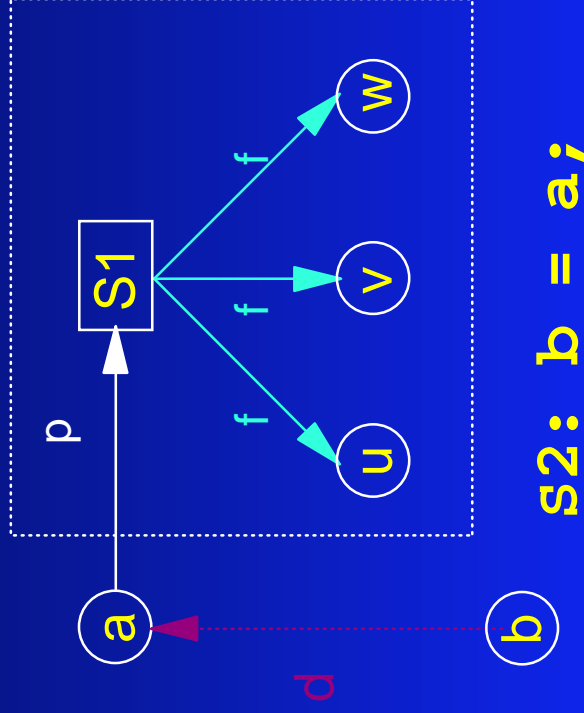
Ref: Choi, Gupta, Serrano, Sreedhar, Mikkiff. Escape analysis for Java. OOPSLA 1999.

Connection Graph

Nodes: objects, object references, fields.

Edges: reachability relationships (points-to, deferred, field).

```
s1: T1 a = new T1(...);
```

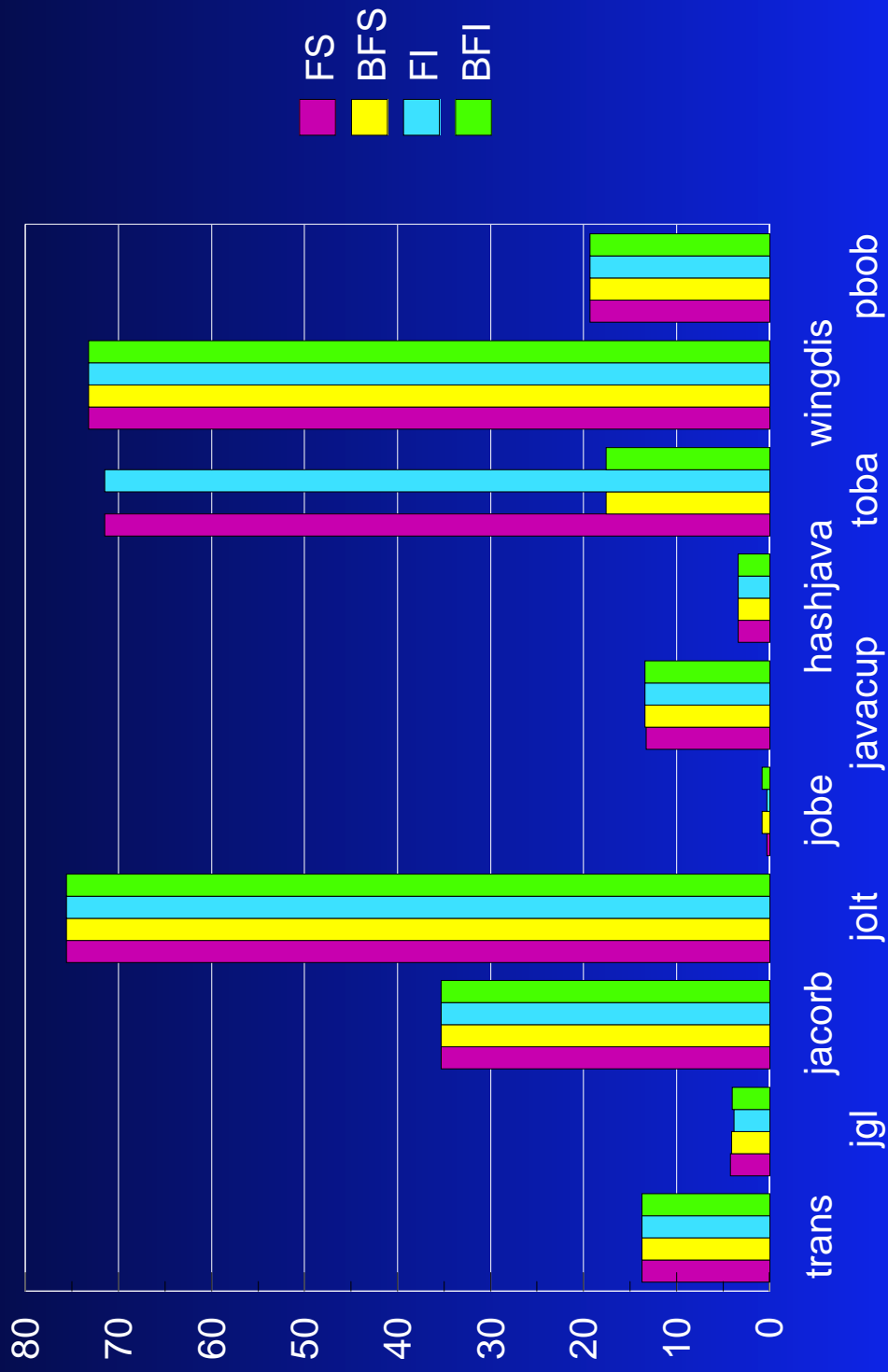


p: points-to edge

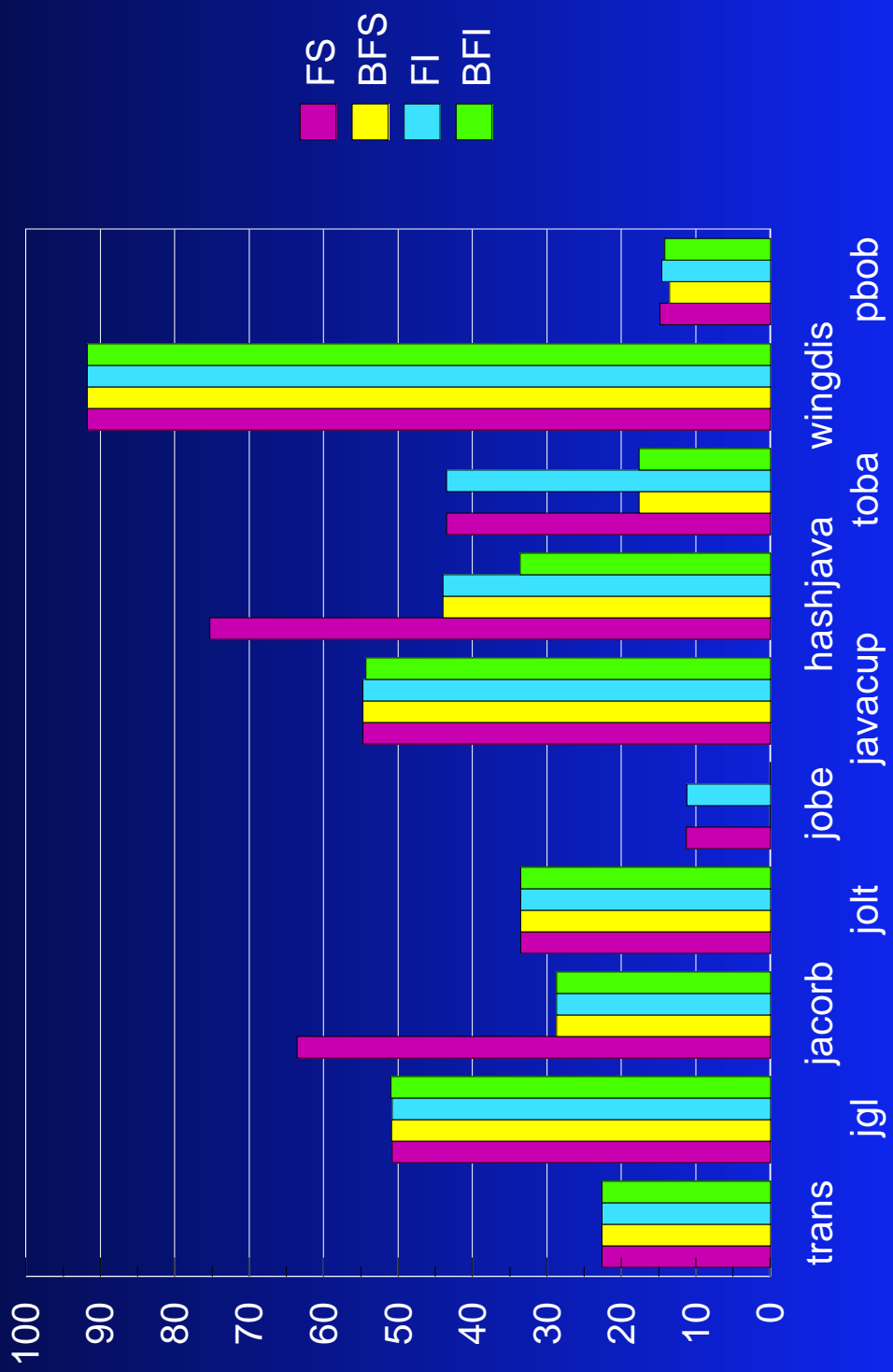
f: field edge

d: deferred edge

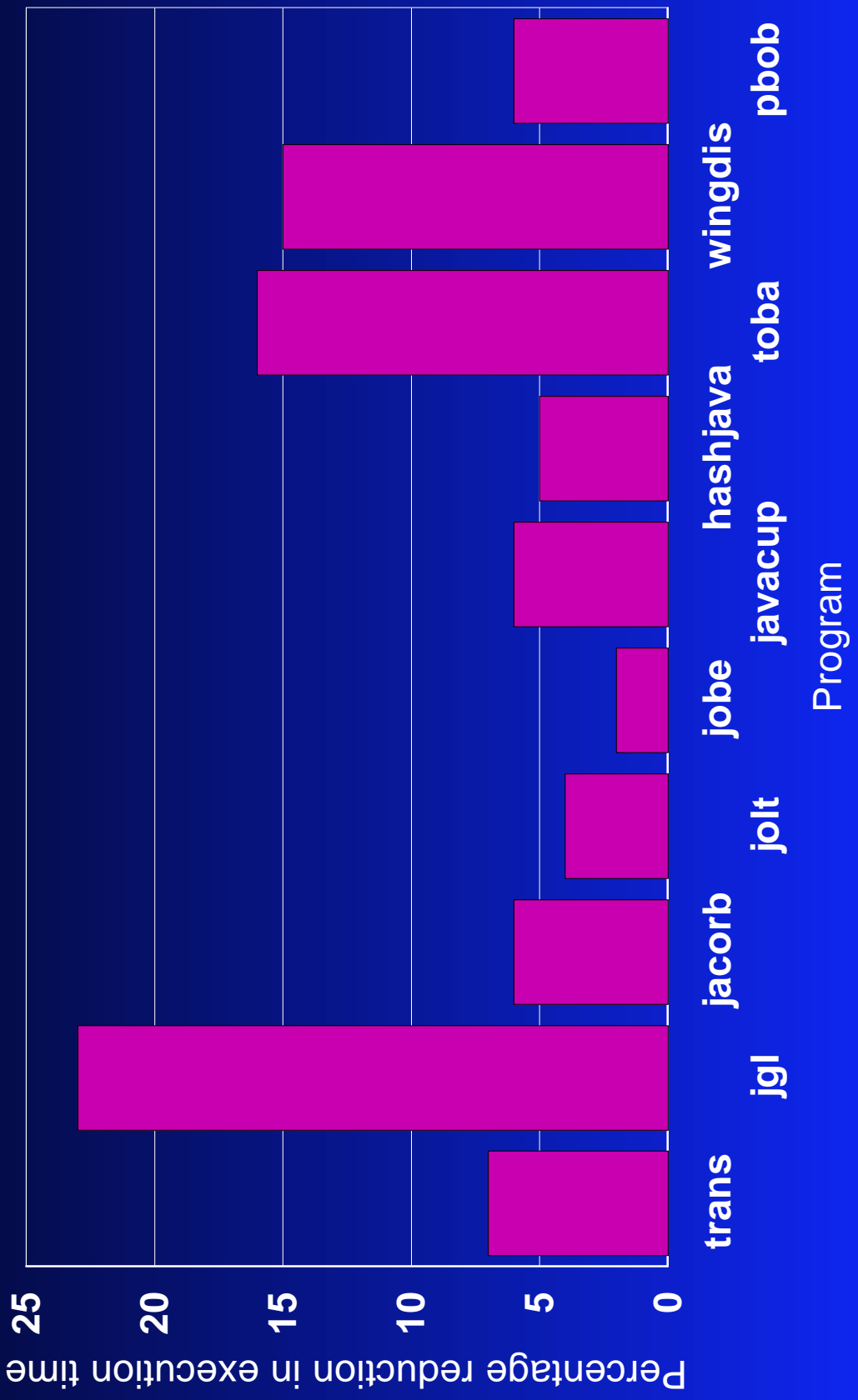
Percentage of User Objects Allocated on Stack



Percentage of Locks Removed on User Objects



Reduction in Execution Time: HPCJ on RS/6000 43P-140



Challenges: Performance Problems

- Inherited from OO programming: virtual method dispatch, space overhead of objects.
- Run-time checks (null pointer, array index out of bounds,...)
- Precise exceptions.
- Automatic memory management: garbage collection.
- Synchronization costs.
- **Dynamic class loading.**
- Memory model too strong.
- Lack of true multidimensional arrays (numerical computing).
- **Run-time binding (affects everything above!).**

Quasi-Static (QS) Compiler: What is it?

- A dynamic (runtime) compiler which generates persistent images of code, and during subsequent executions, reuses those images after suitable adaptation.

Ref: Serrano, Bordawekar, Midkiff, Gupta. Quicksilver: A quasi-static compiler for Java. OOPSLA 2000.

Why not Conventional Static Compilers?

- **Poor match with a dynamic language like Java.**
 - ▶ cannot support true dynamic class loading in Java.
 - ▶ difficulty in coping with late (runtime) binding of method/field references.
- **Inability to perform optimizations across class boundaries (more generally, across DLL boundaries).**
 - ▶ e.g. inlining B.bar() in A.foo() - code for A.foo() is no longer valid if B.bar() has changed.

Why not Conventional Static Compilers?

- **Bad fit with a dynamic language.**
 - ▶ cannot support true dynamic class loading in Java.
 - ▶ difficulty in coping with late (runtime) binding of method/field references.
- **Inability to perform optimizations across class boundaries (more generally, across DLL boundaries).**
 - ▶ e.g. inlining B.bar() in A.foo() - code for A.foo() is no longer valid if B.bar() has changed.
- ▶ **no hope of applying optimizations across the entire middleware stack!**

Advantages over purely Dynamic Compiler

- **Performance benefits**
 - ▶ ability to perform expensive global analysis - amortize cost over many runs.
 - ▶ ability to use training workloads/fixed profiling information to tune programs (more thoroughly).
 - ▶ sharing of compiled library code by multiple JVMs.
 - ▶ smaller memory footprint of compilation at runtime.
- **Reliability and Serviceability benefits**
 - ▶ ability to associate small number of binaries with a source program for reliability, testing purposes.
 - ▶ ability to apply binary patches while JVM is running.

Disadvantages relative to Dynamic Compiler

- Code optimizations less responsive to conditions during current execution.
- No savings (some overhead) if classfiles have changed.
- If classfiles transferred over the network
 - ▶ forgo quasi-static compilation, or
 - ▶ incur overhead of transmitting binary image over network.

Quicksilver: QS Compiler for Java

- **Implementation in Jalapeño VM (PowerPC/AIX).**
- **Generation of QS Images (Compilation)**
 - ▶ binary code, exception tables, GC maps (instance-specific).
 - ▶ relocation and stitching info.
 - ▶ dependence on other classes (digest information).
 - ▶ bytecode verification info.
 - ▶ serviceability info - JVM environment, debugging maps (?).
- **Management of QS Image Repository**
 - ▶ association of QSI with class using digest of class bytecode.
 - ▶ location mapping based on defining class loader.
- **Reuse of QS Images (Execution)**
 - ▶ Check for existence, validity, verification of QS image -- fallback to dynamic compilation, if necessary.
 - ▶ Stitching to adapt QS image to new JVM instance.

Key Challenge: Preserve optimizations across execution instances.

Relocation (Stitching)

- Adapt all instance-specific entries in QS image to the new JVM instance.
- All entries that refer to JTOC elements in Jalapeño VM: need different index into new JTOC.
 - ▶ static fields and methods.
 - ▶ strings, floating point constants.
 - ▶ JVM runtime routines.
- Instance fields and methods.
 - ▶ resolution status (for references to other classes) may be different in different executions.
 - ▶ need stitching because offset may have changed (even for references to same class).

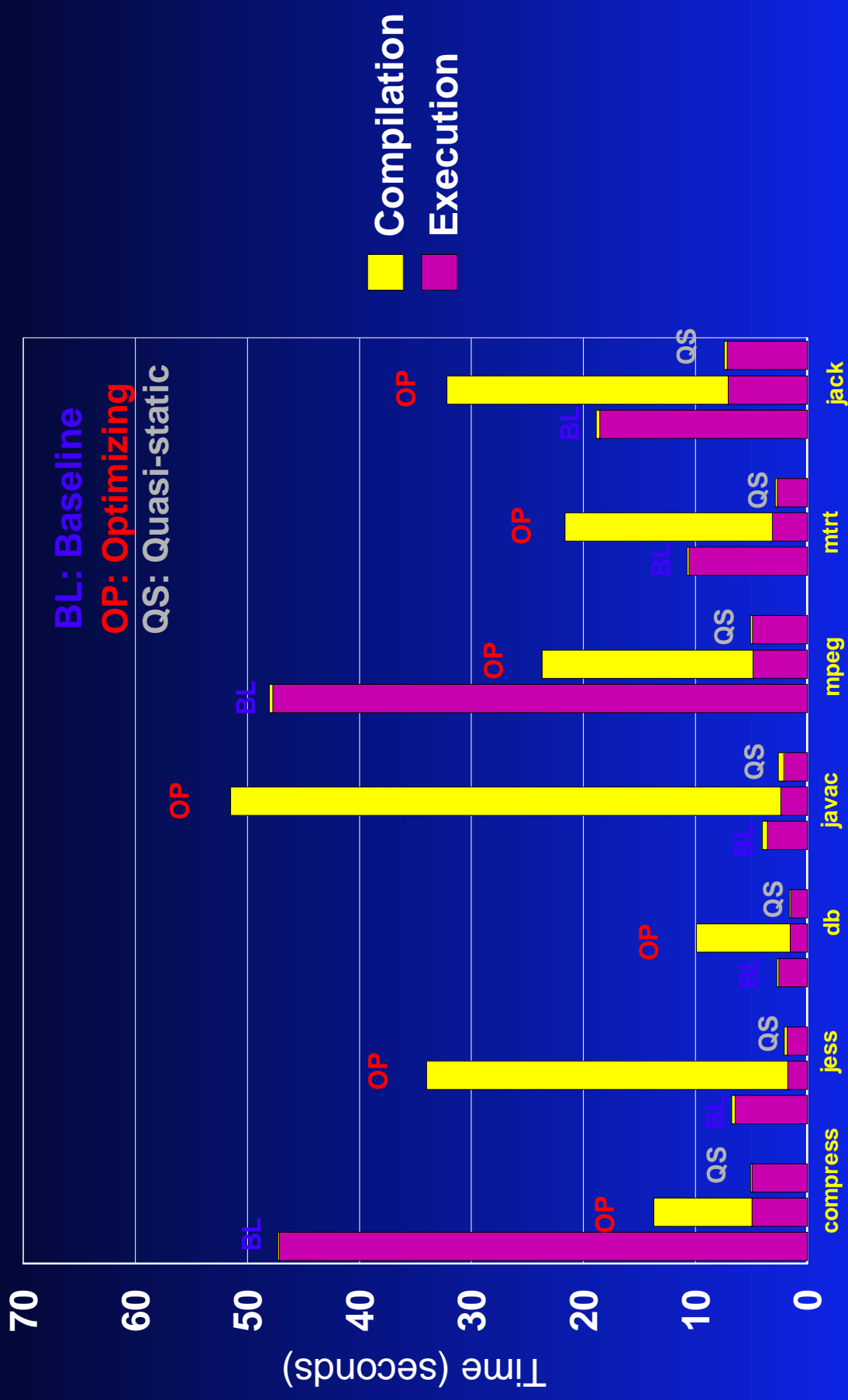
Performance of SPECjvm98 and pBOB

SPECjvm98 size=100, pBOB(4 processors), PowerPC



Performance of SPECjvm98

SPECjvm98 size=10



Benchmarks

Summary of Experimental Results for SPECjvm98

- Factor of 100-150 reduction in runtime compilation overhead relative to Optimizing compiler.
- Relative to **better** of Optimizing and Baseline compilers, overall improvement in performance (considering both compilation and execution time):
 - ▶ 9% - 91% for size 100 (longer running).
 - ▶ 54% - 367% for size 10 (shorter running).

Ongoing Work

Enable optimizations requiring deep analysis.

Conclusions

- Most performance problems of Java are due to features that are good for safety/productivity.
- Rapid progress is being made in improving performance of Java.
- Many Java features hold promise of performance edge in the long run -- strong typing, simpler pointers, flexibility of object layout.