The Design and Implementation of the Jikes RVM Optimizing Compiler

Presenters

David Grove and Michael Hind

IBM Watson Research Center

www.ibm.com/developerworks/oss/jikesrvm

Tutorial Goals

- Educate current or future users of the Jikes RVM optimizing compiler
- Share experiences and perspectives on compiler design and implementation for OO languages

Tutorial Outline

- Background
- Compiler structure
- Selected optimizations
- Compiler/VM interactions
- Perspectives

What is the Jikes RVM?

- Open source version of the code developed by Jalapeño project at IBM Research
- Research virtual machine, not a full JVM™
  - missing libraries (e.g., AWT, Swing, J2EE), JVM protocols (e.g. JVMPI), multiple namespaces for class loaders, "language lawyer" issues, etc.
- Executes Java™ programs typically used in research on fundamental VM design issues
- Provides flexible testbed to prototype new VM technologies and experiment with different design alternatives
- Runs on AIX™/PowerPC™, Linux©/IA-32
  - Linux/PowerPC (with limited functionality/support)
- Industrial-strength performance for many benchmarks

Open Source Highlights

- Announced at OOPSLA (Oct 15, 2001), v2.0.0
  - Minor releases in Nov, Jan, Mar, June, July (v2.0.1, 2.0.2, 2.0.3, 2.1.0, 2.1.1)
  - Accepting contributions since June, 2002
    - 6 contributions in 3 months
- As of Sept 4, 2002 (10+ months)
  - 3,600+ downloads, 1,550+ different sites, 90+ universities, 60+ in US
  - 100+ mailing list subscribers, 900+ messages
- Many users' pubs in top conferences
- Courses at UT-Austin, Wisconsin, UCSB, and New Mexico using Jikes RVM
  - teaching resources available on web site
- Used for a broad range of research topics
  - GC, instruction scheduling, fault tolerant computing, specialization, IR transformations, scheduling multi-threaded apps, OO runtime systems, mobile code security, verification, adaptive optimization, embedded computing, . . .
- Growing Jikes RVM into an *independent* open source project
  - Non-IBM members on Core Team

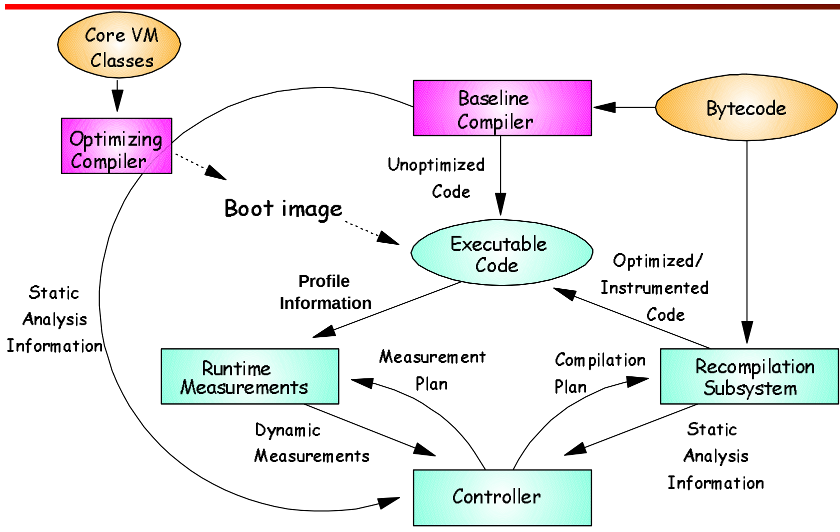
2002 Jikes RVM Users' Pubs

- POPL'02
  - Exploiting Prolific Types for Memory Management and Optimizations by Shuf, Gupta,Bordawekar, and Singh
- SPAA'02
  - The Pensieve Project: A Compiler Infrastructure for Memory Models by C.-L. Wong, Z. Sura, X. Fang, S.P. Midkiff, J. Lee, and D. Padua
- ECOOP'02
  - Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading by Matthew Arnold and Barbara G. Ryder
  - Atomic Instructions in Java by David Hovemeyer, Bill Pugh, and Jamie Spacco
- SIGMETRICS'02
  - Error-Free Garbage Collection Traces: How To Cheat and Not Get Caught by Hertz, Blackburn, Moss, and McKinley
- MSP'02
  - Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine by Stefanovic,Hertz, Blackburn, McKinley and Moss
- PLDI'02
  - Beltway: Getting Around Garbage Collection Gridlock by Blackburn, Jones, McKinley, and Moss
  - Static Load Classification for Improving the Value Predictability of Data-Cache Misses by Burtscher, Diwan, and Hauswirth
  - Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs by Choi, Lee,Loginov, O'Callahan,Sarkar, Sridharan
- LCTES'02/SCOPES'02
  - When to Use a Compilation Service? by Jeffrey Palm, Han Lee, Amer Diwan, and J. Eliot Moss
- ISMM'02
  - In or Out? Putting Write Barriers in Their Place by Steve Blackburn and Kathryn McKinley
  - An Adaptive, Region-based Allocator for Java by Feng Qian and Laurie Hendren
  - Understanding the Connectivity of Heap Objects by Martin Hirzel, Johannes Henkel, Amer Diwan, Michael Hind
- LCPC'02
  - Automatic Implementation of Programming Language Consistency Models by Sura,Wong,Fang, Lee, Midkiff, and Padua
- Java Grande'02
  - Immutability Specification and its Application by Igor Pechtchanski and Vivek Sarkar
- OOPSLA'02
  - Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times by Shuf, Gupta, Franke, Appel, Singh
  - GCspy: An Adaptable Heap Visualisation Framework by Tony Printezis and Richard Jones

Jikes RVM Technical Highlights

- Implemented in Java programming language (~250KLOC)
  - Reduces seams between VM and applications
  - VM can be dynamically optimized
- Compile-only strategy
  - Multiple compilers, mixing code is seamless
- Lightweight (m:n) thread implementation
  - Java threads are multiplexed on OS threads, important for scalability, GC transition
  - Quasi-preemptive scheduling (using compiler-generated yield points)
- Adaptive optimization system
  - Yieldpoint-based sampling, cost/benefit model, what to recompile and what opt. level
  - Online feedback-directed inlining
- Type-accurate (exact) parallel GC/allocation
  - semispace and mark-sweep, generational and nongenerational, hybrids, GCToolkit (UMass)

Compilers in Context



Optimizing Compiler Design Requirements

Input: Bytecode

- No need for lexical analysis, parsing, verification

Output: Machine code + Mapping information [+Analysis results]

Mapping information

GC maps, source code maps, exception tables

Characteristics

- High-quality code generation
- Fast compile-time
- Type-exact GC support
- Support for Java features
  - Exception semantics, dynamic class loading, multithreading, etc.
- Adaptive and feedback-directed optimization

Optimizing Compiler at a Glance

- 3 levels of Intermediate Representation (IR)
  - Java type information preserved
  - Java-specific operators and optimizations
- Multiple optimization levels
  - many classical optimizations
  - some novel optimizations
- Approx. 100K lines machine-independent code
  - 10K lines machine-dependent each for PPC, IA32
- Template-driven generation of instruction formats, command-line arguments, instruction selection, IA32 Assembler
- Interfaces to adaptive optimization system

Tutorial Outline

- ✓ Background
- Compiler structure
  - Intermediate representation
  - Phases
- Selected optimizations
- Compiler/VM interactions
- Perspectives

Intermediate Representation

- "Three-address" like register transfer language
  - Not a stack machine
- Operand 1 ... Operand k = OPERATOR (Operand k+1 ... Operand n)
  - Operands: registers, constants, guards, memory locations, methods, ...

3 levels of Operators

- HIR (High-Level IR)
  - Operators similar to Java bytecode
  - eg. ARRAYLENGTH, NEW, GETFIELD, BOUNDS_CHECK, NULL_CHECK
- LIR (Low-Level IR)
  - Introduces details of Jikes RVM runtime and object layout
    - eg. GET_TIB (vtable), GET_JTOC (static), INT_LOAD (for getfield)
  - Expands complicated HIR operators such as TABLE_SWITCH
- MIR (Machine-Specific IR)
  - Introduces details of target machine --- similar to assembly code
  - Register allocation performed on MIR

HIR Example

Java Source

```
public static void main() {
  System.out.println("Hello world");
}
```

Bytecode

```
Method void main()
  0 getstatic #2 <Field java.io.PrintStream out>
  3 ldc #3 <String "Hello world">
  5 invokevirtual #4 <Method void println(java.lang.String)>
  8 return
```

HIR

```
          LABEL0
     EG   ir_prologue
      G   yieldpoint_prologue
0         getstatic        t0i(java.io.PrintStream,d) = <mem loc: java.lang.System.out>
5    EG   null_check       t1v(GUARD) = t0i(java.io.PrintStream,d)
5    EG   call             LR = <unused>, virtual"java.io.PrintStream.println
                              (Ljava/lang/String;)V", t1v(GUARD), t0i(java.io.PrintStream,d),
                              string constant @12944 JTOC
      G   yieldpoint_epilogue
          return
          bbend            BB0 (ENTRY)
```

E: Potential exception throwing instruction (PEI)
G: Potential garbage collection point

LIR Example

LIR

```
          LABEL0
     EG   ir_prologue
     G    yieldpoint_prologue
0         int_load               t0i(java.io.PrintStream,d) = JTOC(int), 23156
                                      <mem loc: java.lang.System.out>
5    EG   null_check             t1v(GUARD) = t0i(java.io.PrintStream,d)
          materialize_constant   t2i(java.lang.String) = JTOC(int), string constant @129445
5         get_obj_tib            t3i([Ljava.lang.Object;) = t0i(java.io.PrintStream,d), t1v(GUARD)
5         int_load               t4i([I) = t3i([Ljava.lang.Object;), 160
5    EG   call                   LR = t4i([I), virtual"java.io.PrintStream.println
                                    (Ljava/lang/String;)V", t1v(GUARD), t0i(java.io.PrintStream,d),
                                    t2i(java.lang.String) JTOC
     G    yieldpoint_epilogue
          return
          bbend                  BB0 (ENTRY)
```

E: Potential exception throwing instruction (PEI)
G: Potential garbage collection point

MIR Example (PowerPC)

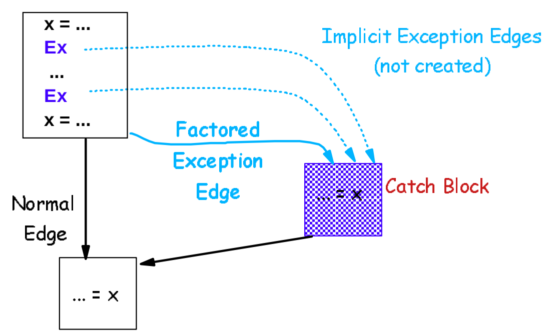
```
            LABEL0
            ppc_mfspr   R0(int) = LR(int)
            ppc_lwz     R13(int) = PR(int), -40
            ppc_stwu    FP(int) <-- FP(int), -16
            ppc_lwz     R14(int) = PR(int), -28
            ppc_lwz     R13(int) = R13(int), -52
            ppc_cmpi    C2(int) = R14(int), 0
            ppc_ldi     R14(int) = 5091
            ppc_stw     R0(int), FP(int), 24
            ppc_stw     R14(int), FP(int), 4
      EG    ppc_tw      ppc trap <, FP(int), R13(int), <STACK OVERFLOW>
      EG    ppc_bcl     LR = C2(int), ppc <, LABEL2 JTOC
   0        ppc_lwz     R3(java.io.PrintStream,d) = JTOC(int), 23156,
                        <mem loc: java.lang.System.out>
   5  EG    ppc_lwz     R4([Ljava.lang.Object;) = R3(java.io.PrintStream,d), -12
   5        ppc_lwz     R4([I) = R4([Ljava.lang.Object;), 160
            ppc_addis   R5(int) = JTOC(int), 1
            ppc_lwz     R5(java.lang.String) = R5(int), 51776, <mem loc: JTOC @51776>
   5        ppc_mtspr   CTR(int) = R4([I)

   ... ETC ....
```

Intermediate Representation

Factored Control Flow Graph (FCFG) [PASTE 99]

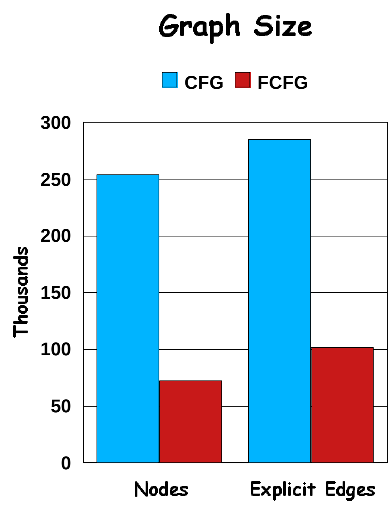
FCFG node
- extended basic block

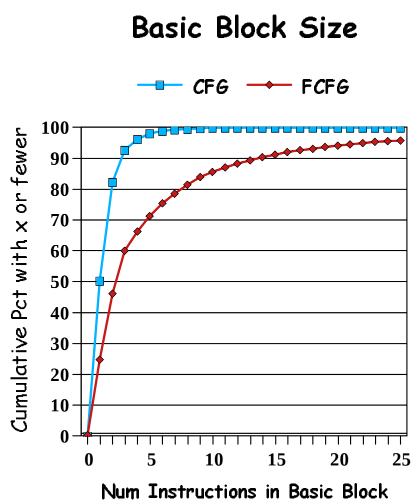
FCFG edge
- normal or exceptional control flow



- reduces number of nodes, edges
- improves scope of local (basic block) analyses
- post-dominance does not hold in a basic block
- straightforward to adapt forward and backward dataflow analyses
- all transformations preserve both normal and exceptional control flow
- approach can be extended to represent superblocks (not currently implemented)

FCFG Size Metrics





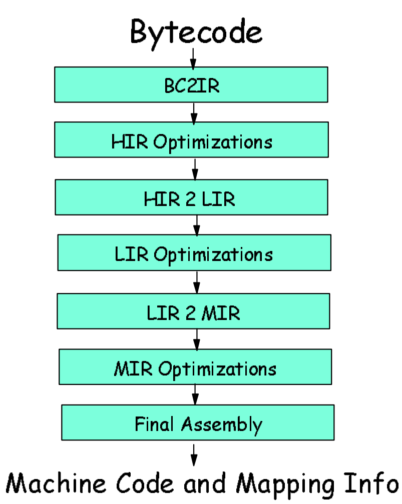
Auxiliary IR Structures

- Computed on demand; not usually maintained
  - Def/Use Chains
  - Dominator Tree, PostDominator Tree
  - Loop Structure Tree
  - Heap Array SSA form
  - Value Numbers / Value Graph
  - Dataflow Equations, Solutions
  - Dependence Graph
  - Interference Graph
  - Register Allocator Analysis Results

Tutorial Outline

- ✓ Background
- ✓ Compiler structure
  - ✓ Intermediate Representation
  - Phases
- Selected optimizations
- Compiler/VM interactions
- Perspectives

Optimizing Compiler Architecture



Each box is composed of multiple compiler phases

HIR (High-Level IR)
- 3-address code
- operations similar to bytecode

LIR (Low-Level IR)
- introduces Jikes RVM details
- (eg. object model, write barriers, etc.)

MIR (Machine-Level IR)
- introduces details of target machine
- similar to assembly code
- register allocation performed on MIR

Implementation Details

A look inside the compiler: Compiler Phases

- Each compiler phase extends OPT_CompilerPhase
- OPT_CompilerPhase.perform: mutates OPT_IR

```
abstract class OPT_CompilerPhase {
  abstract void perform(OPT_IR ir)
}
```

- Compilation: composition of OPT_CompilerPhase.peforms()
- OPT_OptimizationPlanner.java: defines compiler actions as a Vector of OPT_CompilerPhases
- 1. Define a new phase

```
class myPhase extends OPT_CompilerPhase {
   void perform() {
      manipulate the IR ....
   }
}
```

- 2. Add it in appropriate place in OPT_OptimizationPlanner.java:

```
addComponent(/* Vector */ masterPlan, new myPhase());
```

Implementation Details

A Sample Compiler Phase

A Phase to print the IR:

```
class MyIRPrinter extends OPT_CompilerPhase {
  final void perform(OPT_IR ir) {
    System.out.println("START OF IR FOR METHOD " + ir.method);
    for (Enumeration e = ir.forwardInstrEnumerator(); e.hasMoreElements(); ) {
      OPT_Instruction s = (OPT_Instruction)e.nextElement();
      System.out.println(s);
    }
    System.out.println("END OF IR FOR METHOD " + ir.method);
  }
}

addComponent(/* Vector */ masterPlan, new MyIRPrinter());
```

Bytecode to HIR (BC2IR)

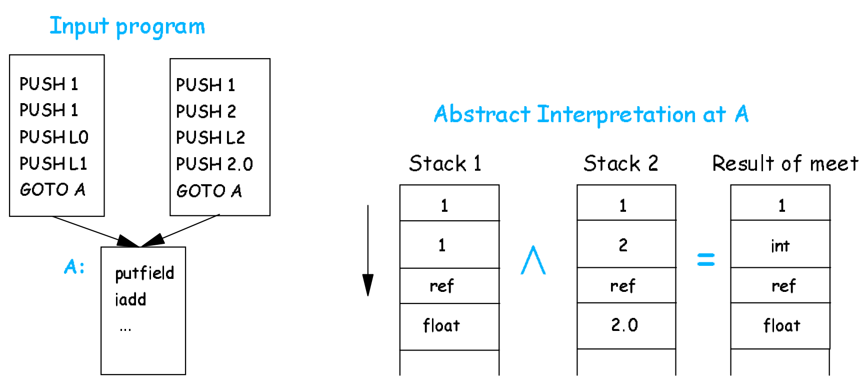
[Java Grande 99]

- Abstract interpretation of bytecodes
- Translates stack instructions to 3-address instructions
- Builds FCFG
- Aggressive, profile-driven interprocedural inlining
- Bytecode subroutines (JSRs) inlined
- On-the-fly dataflow optimizations
  - constant and type propagation, constant folding, branch optimizations, unreachable code elimination
- Yield points inserted after HIR is generated

BC2IR

[Java Grande 99]

- Abstract interpretation of bytecode stream
- Translate stack abstraction to typed register transfer language



HIR to LIR Conversion (HIR2LIR)

- Introduces Jikes RVM details into IR
  - VM services
    - allocation, locks, type checks, write barriers
  - Object model
    - field, array, and static layout; method invocation
  - Other lower-level expansion
    - switches, compare/branch, exception checks

LIR to MIR Conversion (LIR2MIR)

BURS (Bottom-Up Rewrite System)

[Fraser,Hanson,Proebsting 92, Sarkar,Serrano,Simons 01]

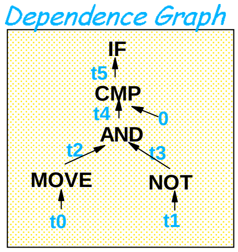
Table-Based Retargetable Instruction Selection

Pattern	Action
reg: REGISTER	<return reg0 = REGISTER>
reg: MOVE(reg)	<return reg0 = reg1>
reg: CMP(AND(reg,NOT(reg))	<emit "andc reg0, reg1, reg2">
stm: IF(reg,!=,LABEL)	<emit "bne LABEL">

LIR

```
t2 := r0
t3 := NOT r1
t4 := t2 AND t3
t5 := CMP t4, 0
if (!t5) GOTO L1
```

Dependence Graph



MIR

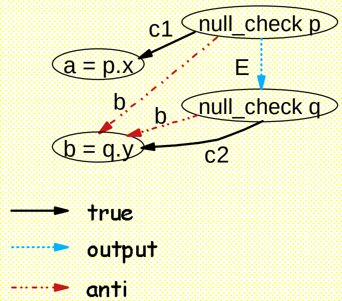
```
andc t5, t0, t1
bne label1
```

Dependence Graph

[LCPC 99]

- Model exceptions, synchronization and memory with aliasing as defs and uses of abstract locations

```
try {                          E,c1 = null_check p [b]
    a = p.x                        a = p.x [c1]
    b = q.y;                   E,c2 = null_check q [b]
                                   b = q.y [c2]
}
catch (NullPointerException e) {
    out.print(b);
}
```



Register Allocation

- Decomposed into machine-independent core and machine dependent utilities
- Basic Linear Scan [TOPLAS 99]
  - single linear-time scan of variable live ranges
  - faster and simpler than graph coloring
- Embellishments
  - live interval holes
  - heuristics to reduce copies
  - smart (?) spill heuristic
  - interface for architecture restrictions (IA32)

Final Assembly

- Machine Code Generation
  - Separate assemblers for PPC, IA32
- GC Maps
- Exception Tables
- Machine Code Info for
  - online profiling, stackframe inspection, debugging, dynamic linking, lazy compilation
  - encodes inlining decisions

Tutorial Outline

✓ Background

✓ Compiler structure

- Selected optimizations
  - Level 0 [Dataflow basics]
  - Level 1 [Flow-insensitive, inlining, commoning]
  - Level 2 [Heap Array SSA, GCP]
- Compiler/VM Interactions
- Perspectives

Standard Optimization Levels

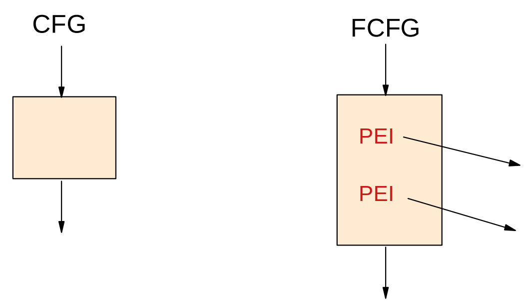
- Level 0
  - On-the-fly constant and type propagation, constant folding, branch optimizations, field analysis, unreachable code elimination, trivial inlining
  - Instruction selection
  - Register allocation and coalescing
- Level 1
  - Full inlining (including preexistence and other speculative inlining)
  - Static splitting, tail recursion elimination
  - Local redundancy elimination (CSE, loads, checks)
  - Flow-Insensitive: constant, copy, type propagation, sync removal, scalar replacement of aggregates, code reordering, dead code elimination
- Level 2
  - Loop normalization & unrolling
  - Scalar SSA: dataflow, global value numbers, global CSE, redundant conditional branch elimination
  - Heap Array SSA: load/store elimination, global code placement

Analysis using the FCFG

[PASTE '99]

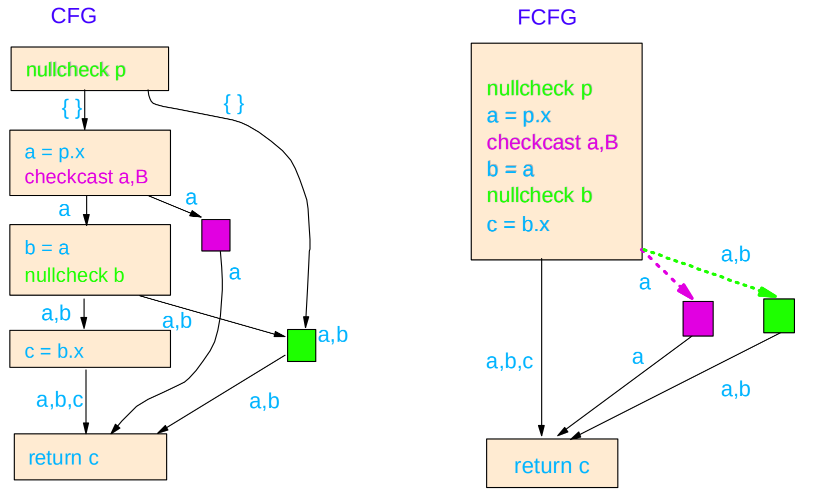
How is analysis performed on the FCFG?

- FCFG does not guarantee *post-dominance* relation among instructions in a basic block



Forward Analysis with FCFG

Reaching Definitions



Forward Analysis Summary

CFG = (N,E)

- Compute & propagate *Gen/Kill* for each basic block

FCFG = (N',E')

- Compute & propagate *Gen/Kill* for each out edge of a basic block

O(|Inst| + k(N+E))

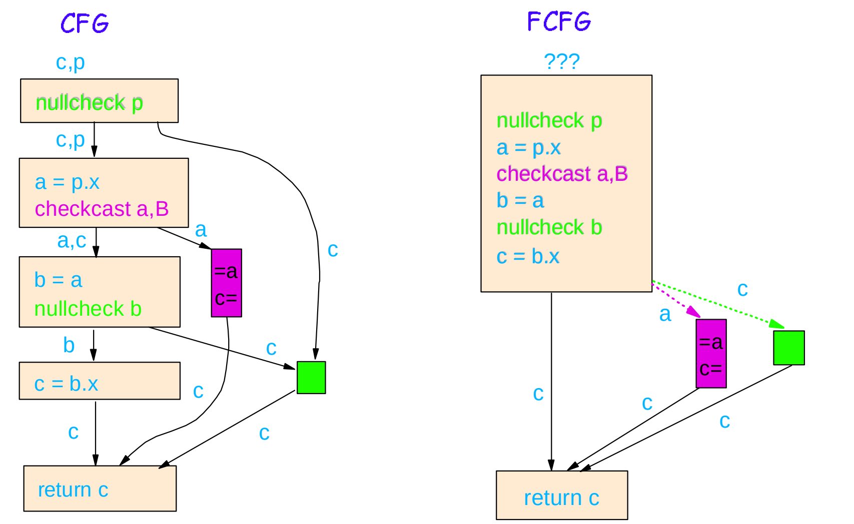
O(|Inst| + k(N'+E'))

N > N', E > E'

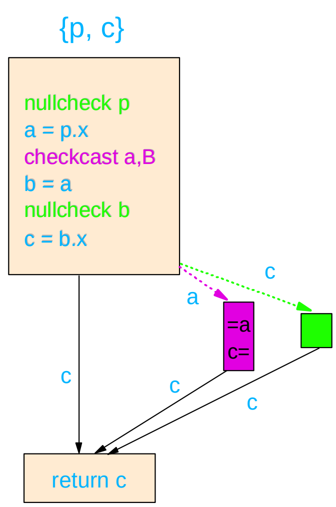
k is height of data flow lattice

No loss of precision compared to CFG

Backward Analysis with FCFG
Live Variable Analysis



Backward Analysis with FCFG
Live Variable Analysis



In = Gen U

(Out - Kill) U

(Out - Kill) U

(Out - Kill)

In = {p} U

{c} - {a,b,c} U

{a} - {a} U

{c} - { }

= {p, c}

Backward Global Analysis Summary

CFG

- Compute *Gen* and *Kill* for each basic block

O(|Inst| + k(N+E))

FCFG

- Compute *Gen* and *Kill* for each basic block
- Compute *Kill* for each *PEI region*

O(|Inst| + k(N'+E'))

N > N', E > E'

k is height of data flow lattice

No loss of precision compared to CFG

FCFG Analysis Summary

[PASTE '99]

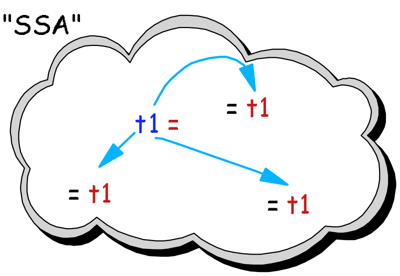
- No loss of precision compared to CFG
- Modifications to CFG-based analysis:
  - Local (within a basic block)
    - Forward: none
    - Backward: minor, at PEIs
  - Global (among basic blocks)
    - Forward/Backward: some
  - Interprocedural (among CFGs)
    - Forward/Backward: "Global" modifications + minor

Dataflow Framework

- Class library for specifying and solving dataflow equations
- Analysis creates an equation system
- Framework reaches sound solution via standard iterative worklist
  - Automatically evaluates in topological order
- Used to perform analysis for load elimination and dead store elimination optimizations

Flow-Insensitive Optimizations

- JLS 4.5.4: "Every variable must have a value before its value is used"
- Easy to identify variables with a single static assignment
- Register-list data structure:
  - track a symbolic reg's defs & uses
  - mark symbolic register with 1 def "SSA"
- fast, conservative versions of
  - dead code elimination
  - copy propagation
  - array bounds checks
  - type propagation



Simple Escape Analysis

- Use register lists to help identify objects that do not *escape*
  (i.e., the object is not live when the current method exits)
- An object **o**, pointed to by a symbolic register **r**, does not escape if
  1. **r** is "SSA",
  2. **r**'s def is a "new" allocation, and
  3. all uses of **r** do not cause **o** to escape
- escape analysis aids optimizations to deal with short-lived objects

Scalar Replacement of Aggregates

- Enabled by escape analysis
- Often applies to Enumerations
- Similar transformation used for small arrays

```
class A {
    int x;
    int y;
}
void foo() {
    A a = new A();
    a.x = 1;
    a.y = a.x + 2;
    System.out.println(a.y);
}
```

→

```
void foo() {
    int t1 = 1;
    int t2 = t1 + 2;
    System.out.println(t2);
}
```

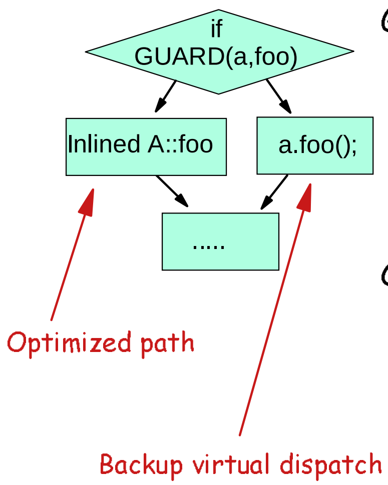
Inlining Mechanism

- Inline any method (via bytecode) into any context
- Occurs during IR generation in concert with on-the-fly optimizations
- BC2IR generates IR into a 'context'
  - enclosing catch blocks
  - initialization of locals (parameters)
  - how to 'return' a value
- Most of IR generation oblivious to inlining

Speculative (Guarded) Inlining

- Guarded inlining of invokevirtual/invokeinterface
- Two reasons for speculation
  - Class Hierarchy Analysis
    - constrained by potential for dynamic class loading
    - guard with class/method test or code patch
    - avoid guards with preexistence
  - Profile-directed
    - Online context-insensitive profile data
    - guard with class/method test

Guarded Inlining Example



Guard implementations

1. class test: class(a) == A
2. method test: a::foo == A::foo
3. code patch: nop/branch

Considerations

- coverage
- runtime cost
- monotonic vs. skewed polymorphic

More on Preexistence

[Detlefs & Agesen '98]

Goal

CHA-based inlining without guards & without requiring on stack replacement on invalidation

```
int foo(A a) {
    ......
    a.m1();
}
```

Key insight

if inlining m1 without a guard is valid when foo is invoked, it will be valid when the inlined code is executed.

Commoning

- Key ideas
  - separate exception checks from computation
  - make as many operations as possible 'ALU' operations:
    - instanceof
    - object-model manipulations
    - exception checks
- Allows standard algorithms for CSE, PRE, etc. to be applied to more interesting computations

Heap Array SSA

[SAS 2000]

- Augments traditional (scalar) SSA transformations
  - Global value numbering
  - Classical forward optimizations
- Extension of type-based alias analysis with SSA flow-sensitivity
- Used for sparse analysis/optimization of heap values
  - Redundant Load Elimination
  - Dead Store Elimination

Heap Array SSA Form

- SSA representation that handles scalars, arrays, and object references in a uniform model
- Three kinds of ϕ functions
  - ϕ : standard control flow merge
  - $d\phi$: definition of memory state
  - $u\phi$: use of a memory state

Array SSA example

[Knobe and Sarkar 98]

Extend scalar SSA form to handle named arrays

```
int A[5];
if (...) {
   A[1] = 1
} else {
   A[1] = 2
}
y = A[1]
```

→

int [] A_1 = new int[5]
if (...) {
  $A_2[1] = 1$
  $A_3 = d\phi(A_1, A_2)$
} else {
  $A_4[1] = 2$
  $A_5 = d\phi(A_1, A_4)$ }
$A_6 = \phi(A_3, A_5)$
$y = A_6[1]$

Heap Arrays

- Model every field with a 1-D Heap Array *x*
  - GETFIELD p.x -> read of *x*[p]
  - PUTFIELD q.x -> write of *x*[q]
- Leverages type system for disambiguation
- Model n-dim array with an n+1-dimensional Heap Array for each array type
  - eg. Java programming language allows only one-dimensional arrays
  - double a = new double [];
  - read (aload) of a[i] -> read of *double* [a, i]

Heap Array SSA example

Introduce "Heap" array *x* for each field x

```
class Z { int x; };
...
Z a = new Z()
if (...) {
   a.x = 1
} else {
   a.x = 2
}
y = a.x
```

class Z { int x; };

...

$Z.x_1[a_1]$ = new Foo()

if (...) {

  $Z.x_2[a_1]$= 1

  $Z.x_3 = d\phi(Z.x_1, Z.x_2)$

} else {

  $Z.x_4[a_1]$= 2

  $Z.x_5 = d\phi(Z.x_1, Z.x_4)$

}

$Z.x_6 = \phi(Z.x_3, Z.x_5)$

y = $Z.x_6[a_1]$

Heap Array SSA Load Elimination Algorithm

1. Build Heap Array SSA form
2. Global Value Numbering (DS & DD)
3. Perform index propagation
4. Scalar replacement analysis
5. Scalar replacement transformation

Load Elimination Example

Original Program

```
p  := new Z
q  := new Z
r  := p
 . . .
p.x  := ...

q.x := ...
 ...  := r.x
```

Transformed Program

```
p  := new Z
q  := new Z
r  := p
 . . .
T1  := ...
p.x  := T1
q.x := ...
 ...  := T1
```

Definitely Same / Definitely Different

- Assign each scalar, s, a value number $\mathcal{V}(s)$
  - Global Value Numbering [AWZ 88]
- Definitely Same (DS)
  - if $\mathcal{V}(x) = \mathcal{V}(y)$, x and y have the same value wherever both are defined
- Definitely Different (DD): harder to compute
  - pointers from different allocation sites
  - preexisting objects [Detlefs & Agesen 99]
  - integers from data dependence analysis

Index Propagation Example

Compute $\mathcal{L}(H)$ = {v ∈ Value Numbers | $H[v]$ is available}

Heap Array SSA representation

p := new Z
q := new Z
r := p
$Z.x_1$ [p] := ...
$Z.x_2 = d\phi(Z.x_0, Z.x_1)$
$Z.x_3$ [q] := ...
$Z.x_4 = d\phi(Z.x_2, Z.x_3)$
... = $Z.x_4$ [r]
$Z.x_5 = u\phi(Z.x_3, Z.x_4)$

Dataflow Solution

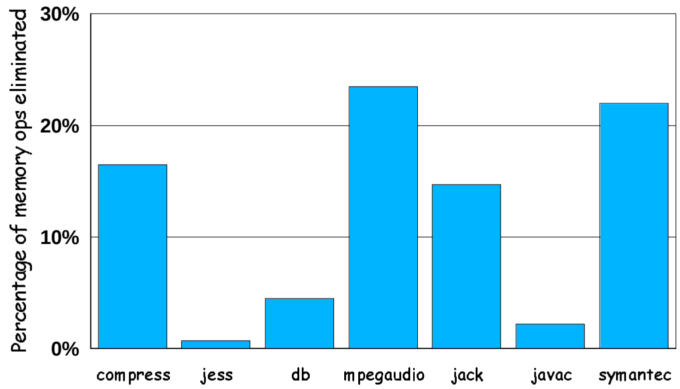
DD (p,q) = true

DS (p,r) = true

- $\mathcal{L}(Z.x_0) = \{\}$
- $\mathcal{L}(Z.x_1) = \{\mathcal{V}(p)\}$
- $\mathcal{L}(Z.x_2) = \{\mathcal{V}(p)\}$
- $\mathcal{L}(Z.x_3) = \{\mathcal{V}(q)\}$
- $\mathcal{L}(Z.x_4) = \{\mathcal{V}(p), \mathcal{V}(q)\}$
- $\mathcal{L}(Z.x_5) = \{\mathcal{V}(p), \mathcal{V}(q)\}$

Dynamic Results

[SAS '00]



(memory op = getfield, putfield, getstatic, putstatic, aload, or astore)

Loop Normalization

Increase opportunities for Code placement

- Partial Loop Peeling: Replicate initial segment of loops
  - Increases code motion opportunities for instructions that must not be executed speculatively (Stores, PEIs)
  - Subsumes while -> until transformation
- Loop Peeling: Prepend loops with a copy of the loop body
  - Renders PEIs in the loop body as non-exceptional instructions if dominated by their copy
  - Makes irreducible loops reducible
- Landing Pad Insertion: Insert new blocks on critical edges
  - Only between blocks with very different execution frequencies

Loop Unrolling

Two strategies: Counted Loops and Naive Unrolling

- Unrolling of counted loops. No loop exits between copies:
  - Pattern matching to identify counted loops
  - Insert pre loop to align iterations with unroll factor
  - Replicate loop body n times and run with an n times larger stride
- Naive Unrolling. Loop can exit between copies:
  - Generally applicable for any loop
  - Used as a fall back if unrolling for counted loops is not applicable
- Aggressiveness controlled by adaptive system
  - Unroll more loops in hot methods

Global Common Subexpression Elimination

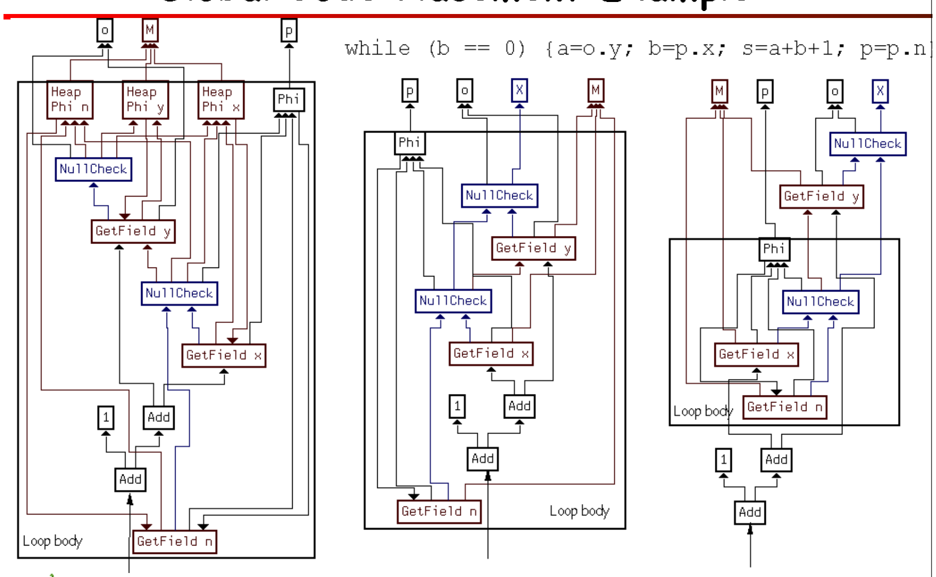
- Walks the dominator tree to eliminate fully redundant instructions
- Uses global value numbering to determine semantically equivalent computations
- If the same value is computed multiple times on a path in the dominator tree, the result of the first definition is reused instead of being recomputed

Global Code Placement

Find a less frequent block for an instruction

- Subsumes loop invariant code motion
- Runs on HIR and LIR using dynamic feedback (block frequencies)
- Views Heap SSA Form as a dependence graph
- Augments the graph with ***dependencies that model Java's ordering constraints*** for loads, stores, and PEIs, and prunes non-essential and not-observable dependencies
- Uses scalar, heap, and exception dependencies to determines earliest and latest positions for instructions
- Searches the dominator path between earliest and latest for a target block with minimal execution frequency

Global Code Placement Example



Heap SSA   Augmented & Pruned   Final Placement

Global Code Placement Algorithm

Earliest (inst)	=	original block of inst, if inst can't be moved
	else	select (Def(inst), inst)
Def (inst)	=	{ Earliest (dep) | inst depends on dep}
speculative		select (B,inst) = b in B with largest dominator depth
non speculative		select (B,inst) = earliest x that is dominated by all b in B and is post-dominated by inst.
Latest (inst)	=	original block of inst, if inst can't be moved
	else	common dominator (Use (inst))
Use (inst)	=	{ finalPos (dep) | dep depends on inst}
finalPos (inst)	=	node with minimal execution frequency on dominator tree path from Earliest (inst) to Latest (inst).

Tutorial Outline

- ✓ Background
- ✓ Compiler structure
- ✓ Selected optimizations
- Compiler/VM interactions
  - Compilation and support of runtime services
  - Adaptive optimization system (AOS)
  - Support for speculative optimizations
- Perspectives

Inlining of Runtime Services: Allocation (1)

HIR:

```
t1 = new java.lang.Object
return t1
```

HIR2LIR implements "new" by calling VM_Runtime.quickNewScalar

```
public static Object quickNewScalar(int size, Object[] tib, boolean hasFinalizer)
   throws OutOfMemoryError {
   Object ret = VM_Allocator.allocateScalar(size, tib);
   if (hasFinalizer) VM_Finalizer.addElement(ret)
   return ret;
}
```

Semispace version of VM_Allocator.allocateScalar

```
public static Object allocateScalar (int size, Object[] tib)
    throws OutOfMemoryError {
    VM_Address region = getHeapSpaceFast(size);
    Object newObj = VM_ObjectModel.initializeScalar(region, tib, size);
    return newObj;
}
```

Inlining of Runtime Services: Allocation (2)

compiler inlines "hot paths", resulting in (optimized) LIR:

```
get_class_tib        t6si([Ljava.lang.Object;) = java.lang.Object
int_load             l23si(VM_Address) = PR(VM_Processor), -36
int_add              l25si(VM_Address) = l23si(VM_Address,d,p), 8
int_load             t30si(VM_Address,d,p) = PR(VM_Processor), -40
boolean_cmp          t31si(int) = l25si(VM_Address), t30si(VM_Address), <=U
int_ifcmp            t31si(int), 0, ==, LABEL3, Probability: 0.5
LABEL1
int_store            l25pi(VM_Address), PR(VM_Processor), -36
LABEL2
int_add              t45si(VM_Address) = l25pi(VM_Address), 12
int_store            t6si([Ljava.lang.Object;), t45si(java.lang.Object), -12
return               t45si(java.lang.Object)
LABEL3
ref_load             t62si([I) = JTOC(VM_Address), 20616, <mem loc: JTOC @20616>
call                 l25pi(VM_Address) LR = t62si([I), static"VM_Chunk.slowPath1
                          (I)LVM_Address;", <unused>, 8 JTOC
goto                 LABEL2
```

Dynamic Type Checking

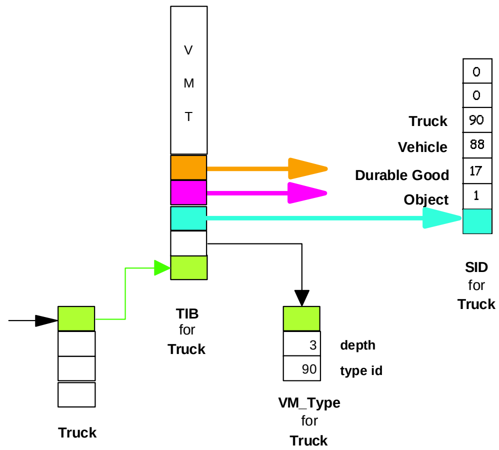
[JVM'01]

- Support instanceof, checkcast, invokeinterface, and aastore bytecodes
- Key ideas:
  - exploit compile-time knowlege to customize dynamic type checking code sequence
  - co-design of VM data structures & inline opt code
  - short inline sequences for common case; uncommon case handled by out-of-line VM routines

Testing for a Proper Class

- A is known to be a proper class (ie not an interface)
  - checkcast and instanceof bytecodes
  - Most significant case (along with aastore)
- Superclass Identifier Display (SID) [Cohen '91]
  - A class's display contains its type id and the type ids of its ancestors
  - The display is ordered (indexed) by their depth
- Dynamic type check:
  - Compare depth SID entry of object to type id of A
  - if A_depth >= minimum (6) then array bounds check required

Superclass Identifier Display (SID)



Dynamic type check:

```
l    r1, TIBoffset(b)
l    r1, SIDoffset(r1)
l    r1, A_depth<<1(r1)
cmpi r1, A_id
bne  NoMatch
```

Dynamic type check:

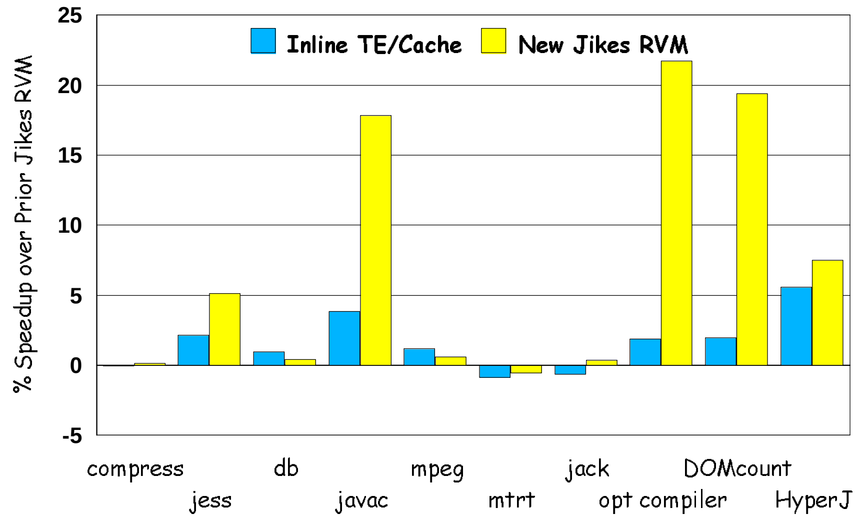
```
l    r1, TIBoffset(b)
l    r1, SIDoffset(r1)
l    r2, lengthOffset(r1)
cmpi r2, A_depth
bge  NoMatch
l    r1, A_depth<<1(r1)
cmpi r1, A_id
bne  NoMatch
```

- Each type has a depth and a type id
- SID for a type is an array of shorts
  - Maps superclass depth to superclass type id
  - Padded to a minimum depth with invalid ids

Experimental Evaluation

- Three dynamic type checking schemes implemented
  1. Prior Jikes RVM: out-of-line type equality, cache last success
  2. Inline TE/Cache: inlined type equality and type cache check
     approximate IBM SDK [Ishizaki et al.'00]
  3. New Jikes RVM [JVM '01]
- All three include same basic optimizations
  1. final classes & null handled inline
  2. null and type propagation
  3. compile time folding of instanceof & checkcast
- Experimental setup
  1. AIX/PowerPC, 1 processor 604e with 768MB
  2. Jikes RVM adaptive system
  3. copying, non-generational garbage collector

Performance Impact of Dynamic Type Checking



Space Considerations

- Data Structures (per ***Type*** costs) are modest
- Code costs are highly variable
  1. Which sequences are inlined?
  2. Where are the sequences inlined (be selective based on profiling)?
  3. Some sequences are both ***definitive*** and ***smaller*** than calls to runtime
     a. Type equality test in restricted cases
     b. Proper subclass using SID
     c. A few array cases

Support for Runtime Services

- Compilers must generate more than just code
- Map from machine code to original bytecodes.
  - Required at PEI's and GC points, optionally others
  - Includes encoding of inlining structure to present view of virtual call stack
  - Used to support dynamic linking, lazy compilation, source level debugging (accurate stack traces), ...
- Exception tables
- GC Maps (next slide)
- Total space cost is 2/3 of machine code (IA32)

Exact GC Support

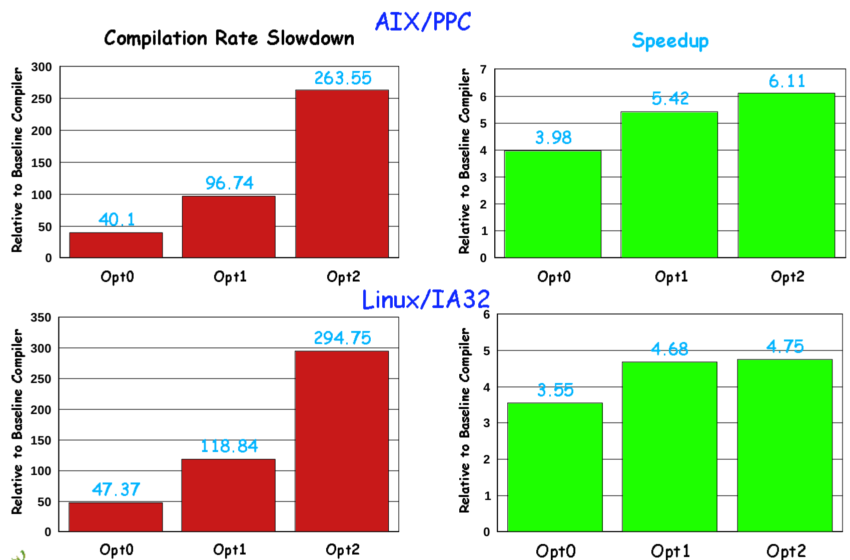
- Compiler generates GC Map for every program point where GC may occur
- GC Map indicates exactly which physical registers and stack locations hold object references
- GC Map Generation
  - All optimizing compiler phases preserve type information until register allocation

  1. Before register allocation:
     - Liveness analysis: backwards dataflow to determine live references
  2. After register allocation:
     - Record register or stack location for each symbolic live reference
  3. After final assembly:
     - Encode GC map compactly and map to machine code offset

Tutorial Outline

- ✓ Background
- ✓ Compiler structure
- ✓ Selected optimizations
- Compiler/VM Interations
  - ✓ Compilation and support of runtime services
  - Adaptive optimization system (AOS)
  - Support for speculative optimizations
- Perspectives

Compiler Performance (Sep'02)

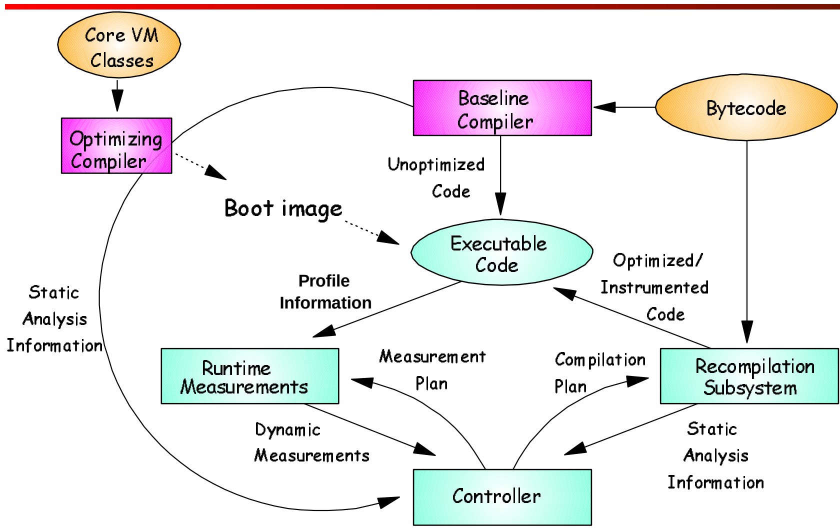


Adaptive Optimization System (AOS)

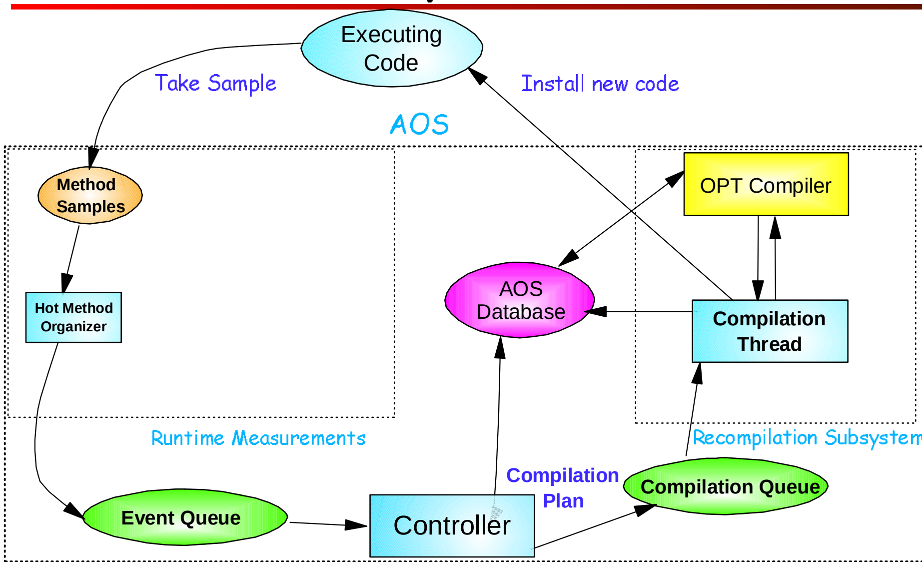
[OOPSLA'00]

- **Goal**: provide an extensible infrastructure to support research in adaptive optimizations
- **Main components** (separate Java threads)
  - runtime measurements
  - controller
  - recompilation subsystem
- **Characteristics**
  - lower overhead sampling occurs throughout execution
    - no invocation counters
  - all methods are initially baseline compiled
  - optimizing compiler (at various levels) used to recompile a subset of methods
  - online system, exploits "characteristics" of current run

Architecture



Basic Implementation



Runtime Measurements

- Samples occur at *taken* yield points
  - infrequent (approximately 100/sec)
  - coarse-grained, low overhead
  - Samples *can* occur at
    - method prologues, epilogues, and loop back edges
- *Organizer* thread communicates sampled methods to controller
  - all methods that are samples in the recent interval

Cost/Benefit Model

- Choose $j > cur$ that minimizes $T_j + C_j$

  cur, current opt level for method *m*

  Tj, expected future execution time at level *j*

  Cj, compilation cost at opt level *j*
- If $T_j + C_j < T_{cur}$ recompile at level *j*
- Assumptions
  - Method will execute for twice its current duration
  - Compilation cost and speedup are offline averages
  - Sample data determines how long a method has executed

Multi-Level Adaptive Performance

(without FDO)

5 configurations (4 non-Adaptive, 1 Adaptive)

- Baseline
- Opt 0
- Opt 1
- Opt 2

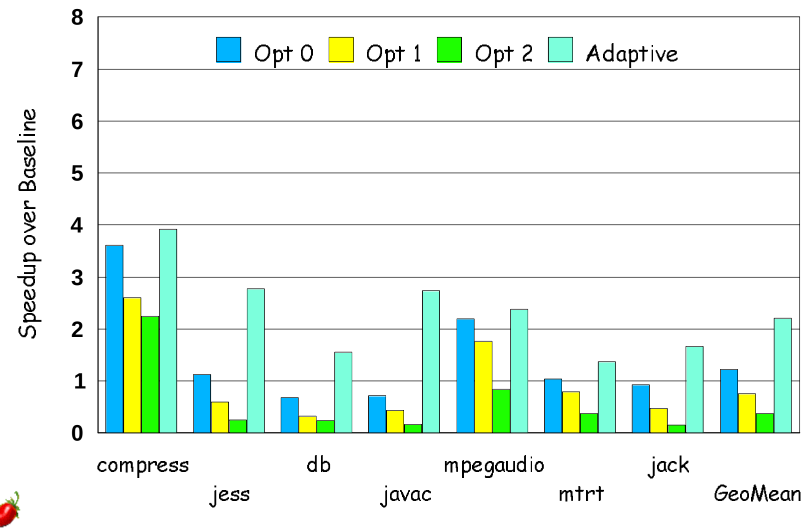
} compile all methods when first called with appropriate compiler

- Adaptive, baseline compile, use cost/benefit model for optimization

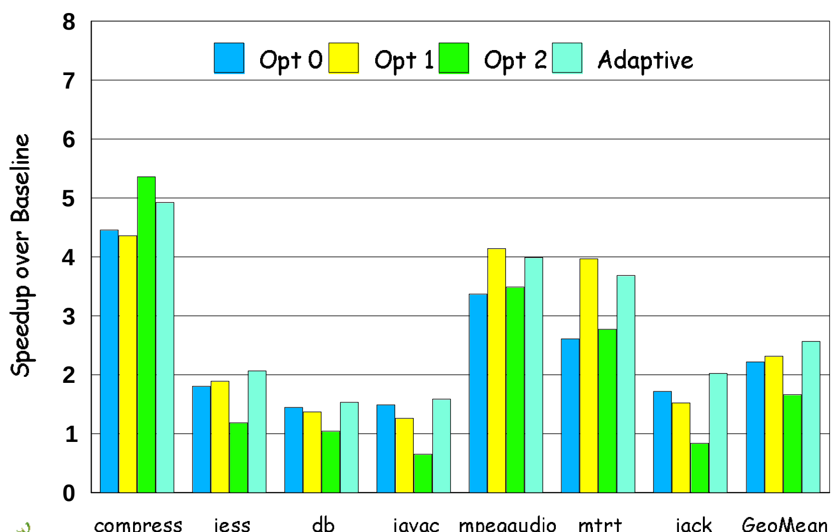
Experiments

- SPECjvm98, 1 processor, AIX/PPC
- Regimes
  - First run (size 10, 100)
  - Best run of 20 runs (size 100)
- *Total time*
  - includes all compilation, profiling, and execution time

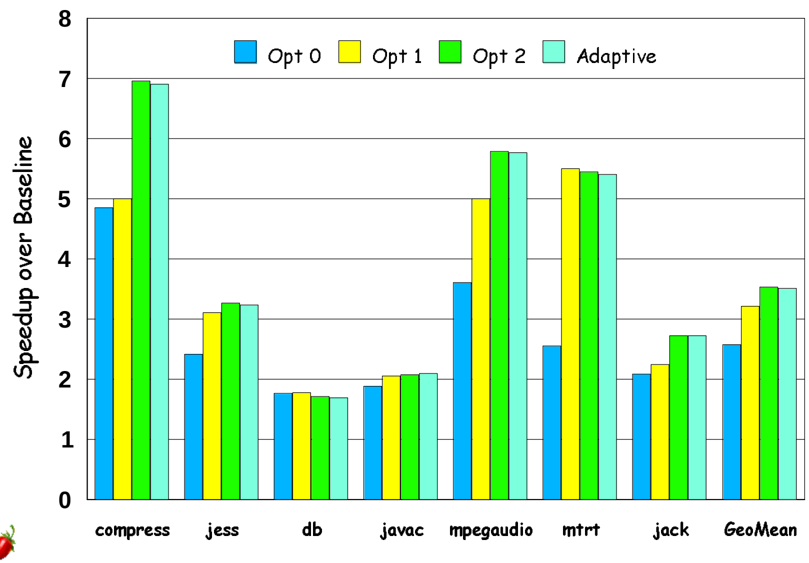
First Run - Size 10



First Run - Size 100



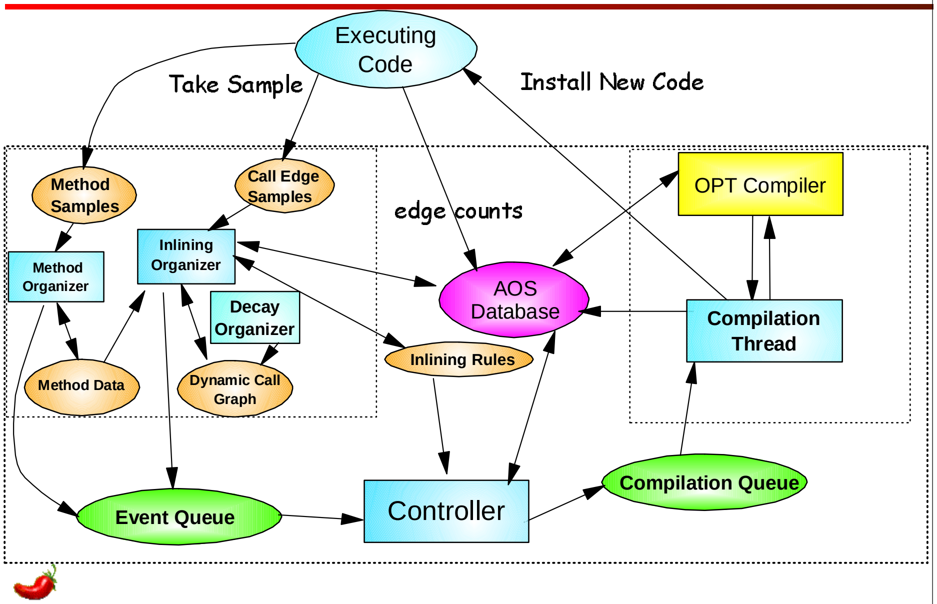
Best of 20 Runs (1 VM)



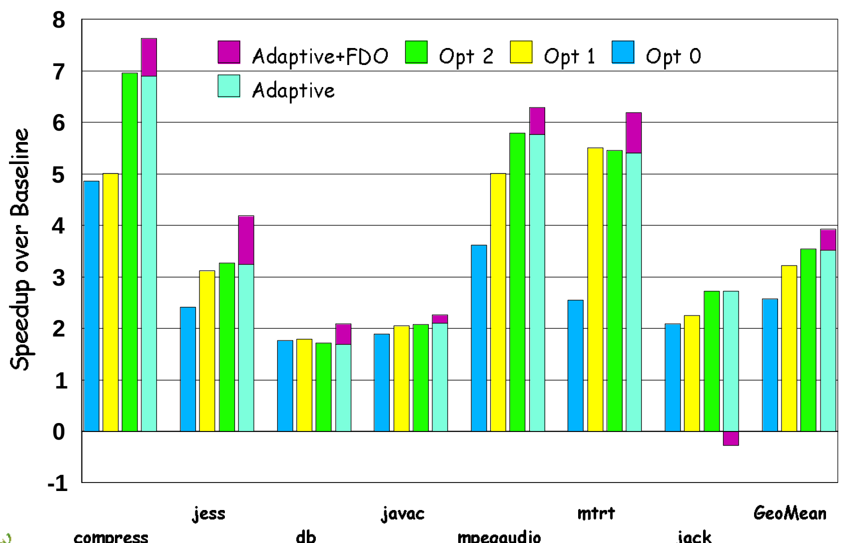
Feedback-Directed Optimizations (FDO)

- Adaptive Inlining
  - Hot call edges identified by prologue samples
    - same low overhead mechanism
    - samples are decayed
  - Inline a method if
    - static size-based heuristics are satisfied, or
    - call edge is hot
- CFG Edge Profiles
  - Collected during baseline-compiled execution
  - Used by opt compiler
    - register allocator
    - loop unrolling
    - code placement
    - code layout
    - out-of-SSA translation

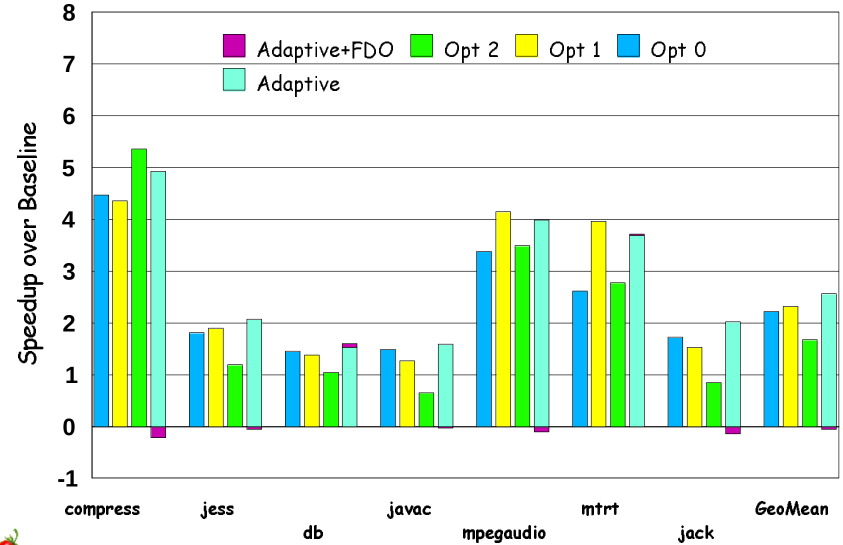
AOS with FDO



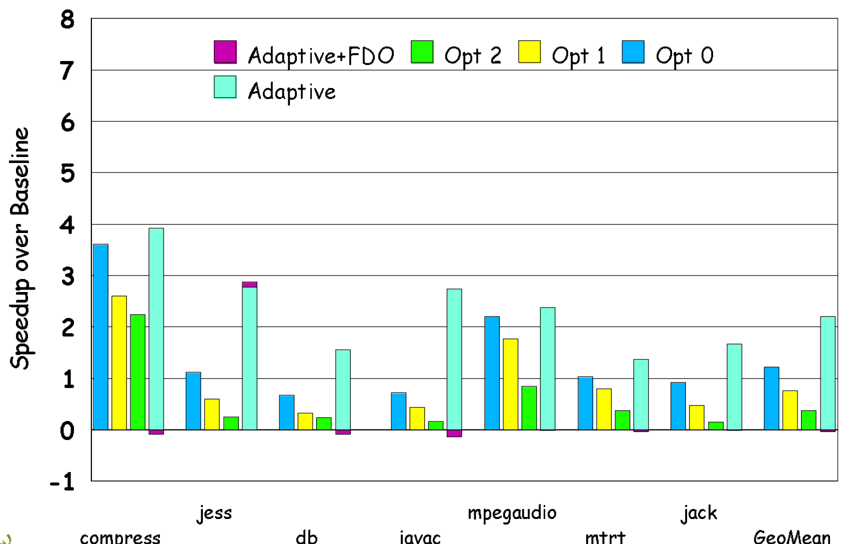
Best of 20 Runs - Size 100



First Run - Size 100

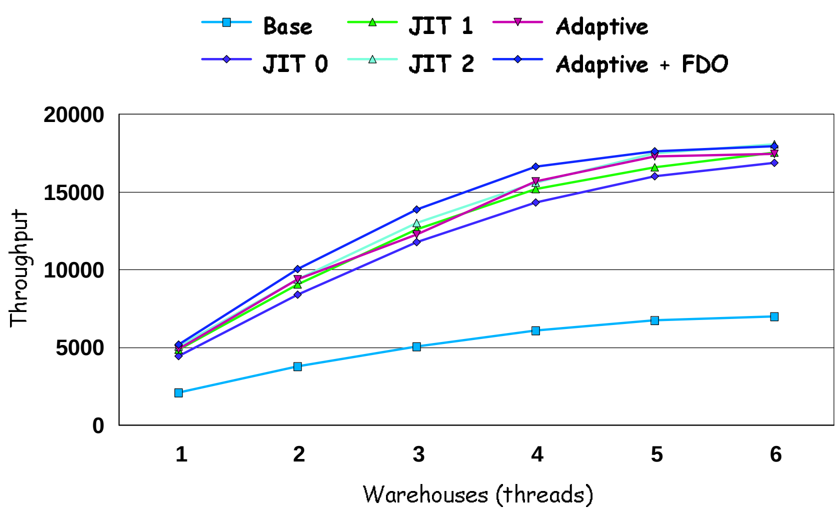


First Run - Size 10



SPECjbb2000

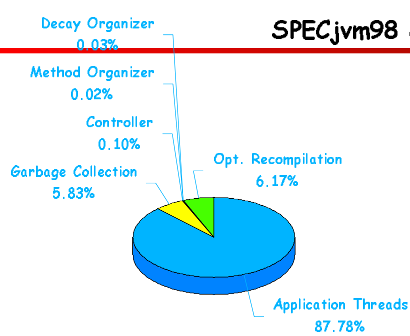
6 Processors

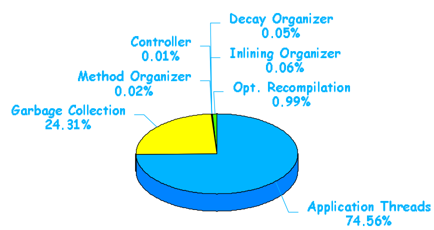


AOS and Jikes RVM's MP load balancer interact badly at full load

Execution Time

SPECjvm98 & SPECjbb2000





SPECjvm98

0.23% AOS overhead

6.17% recompilation

Size 100, 1 run each

SPECjbb2000

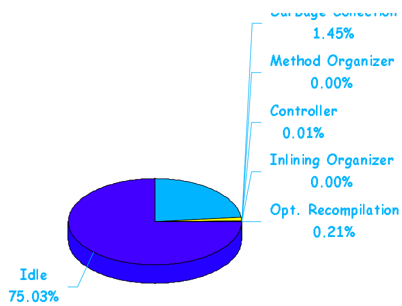
0.14% AOS overhead

0.99% recompilation

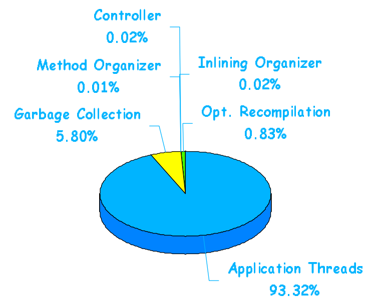
1..8 warehouses, 6 Procs

Execution Time

Eclipse



Ignoring Idle Time

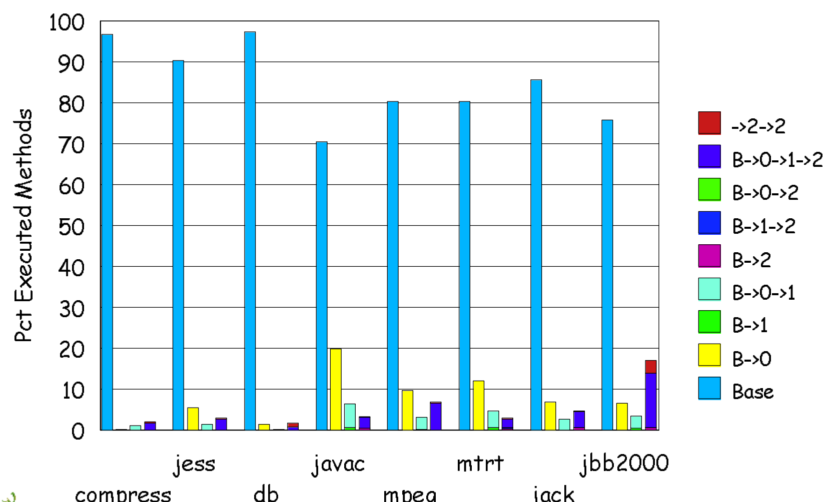


0.01% AOS overhead
0.21% recompilation
75% idle time

0.06% AOS overhead
0.83% recompilation

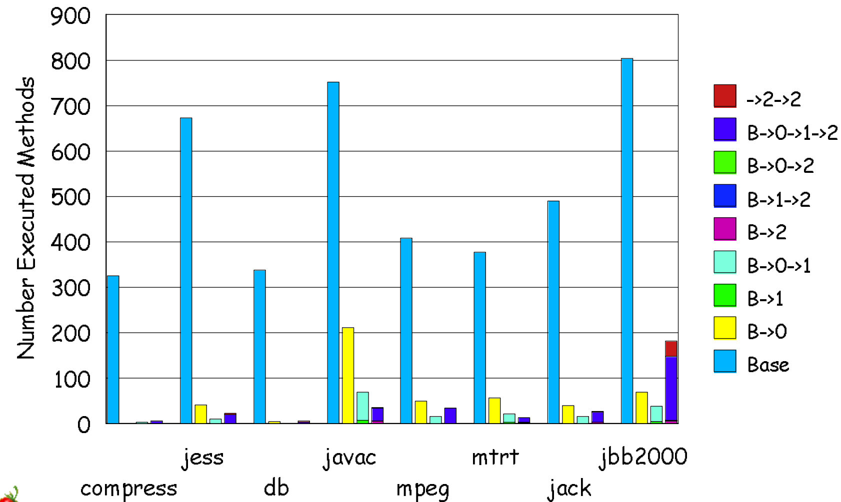
Recompilation Decisions - PCT

SPECjvm98: 20 Runs, SPECjbb2000: 1..8 warehouses, separate VMs



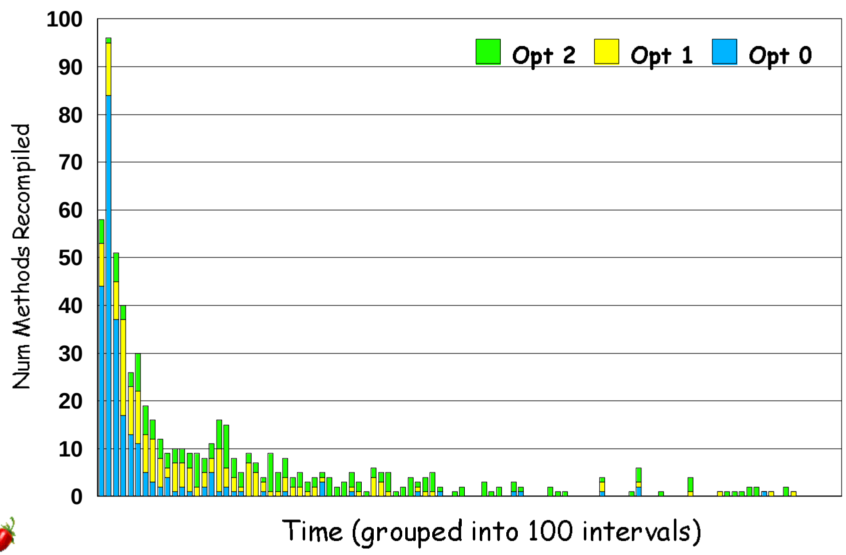
Recompilation Decisions - Num

SPECjvm98: 20 Runs, SPECjbb2000: 1..8 warehouses, separate VMs



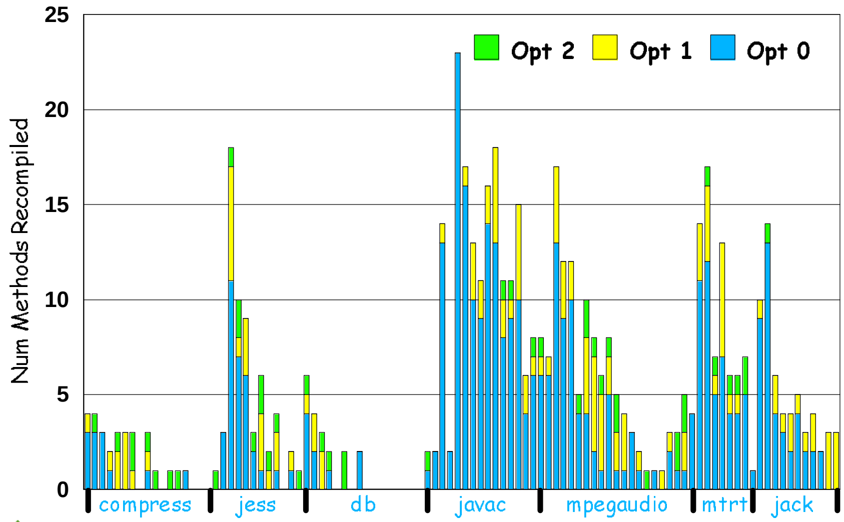
Recompilation Activity

SPECjbb2000, 1..8 warehouses, 6 procs



Recompilation Activity

1 run of each benchmark in same VM instance



AOS Lessons Learned

- AOS is distributed, asynchronous, and object-oriented
  - allows managing data efficiently
  - modularity allows extensible architecture
- Sample-based profiling allows adaptive adjustment of overhead without recompiling
- Analytic model better than ad-hoc tuning parameters
- Programming in Java
  - reduced implementation and debugging time
  - supported asynchronous design

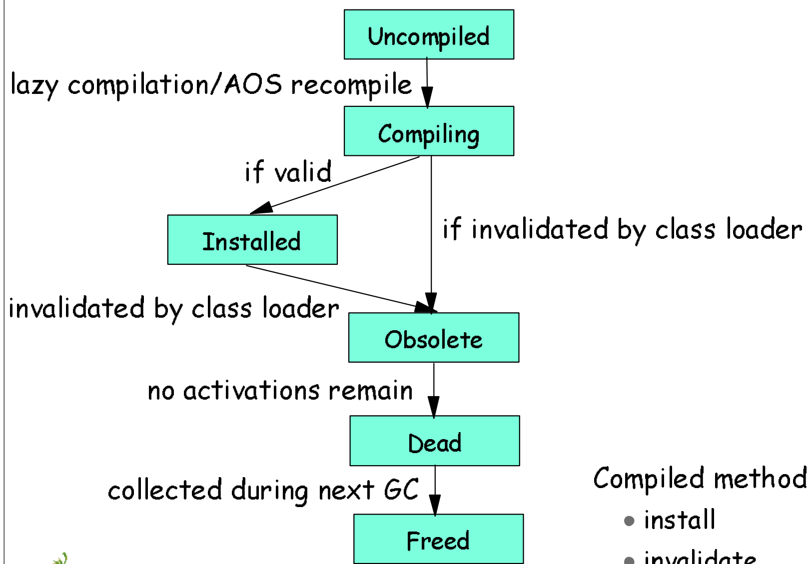
Tutorial Outline

- ✓ Background
- ✓ Compiler structure
- ✓ Selected optimizations
- Compiler/VM Interations
  - ✓ Compilation of runtime services
  - ✓ Adaptive optimization system (AOS)
  - Support for speculative optimizations
- Perspectives

Support for Speculative Optimizations

- Compiler optimizes based on currently loaded program (optimistic)
- Dynamic class loading can invalidate previous optimistic assumptions
- Invalidation database records dependency of compiled code on particular properties being true
- As classes are loaded, dependencies are checked and compiled code may be invalidated
- Fine grained locking to minimize synchronization
  - enable class loading & compilation to occur in parallel

States of Compiled Methods



Compiled method locked to

- install
- invalidate

Invalidation Mechanisms

- Invalidate code by replacing TIB/JTOC entries by "lazy compilation stub"
  - method will be recompiled on next invocation
- GC and compiled code manager coordinate to detect/collect obsolete code
  - code manager keeps set of obsolete code
  - GC stack scanning marks code as active
  - obsolete, inactive code collected by next GC

Tutorial Outline

- ✓ Background
- ✓ Compiler structure and phases
- ✓ Selected optimizations
- ✓ Compiler/VM interactions
- Perspectives

Writing VM/Compiler in Java Programming Language

- High-level (compared to C) features suit compiler development
  - strong typing
  - automatic memory management
  - no buffer overflows
- java.util.* is useful . . . most of the time (beware of equals())
- Javadoc useful
- Lack of parametic types results in many downcasts (generics please!)
- Cpp-style preprocessor helps in many cases
- Good to write in language you are compiling ("eat your own dogfood")
- Avoided most "endian" porting issues
- Separating compilation from benchmark performance: open issue
- Jikes RVM proves you can write a VM in Java (+magic) that can compete with the best VMs written in C.

Lessons Learned

- Control over both runtime and compiler invaluable
  - Compiler-runtime coordination key for modern language features
  - eg. type tests, method dispatch, synchronization, GC, etc.
- Mechanical code generation avoids errors and tedious coding
  - assembler, operators, options
- Nightly regression tests
  - wish we had started these earlier
- Bug and feature tracking software
- Debugger
  - wish we'd devoted more effort to debugger earlier

IR Design Issues

- IR heart of compiler: redesign expensive & disruptive
- Using same IR format for all IR levels allows reuse
  - many analyses/optimizations run unchanged on multiple levels
  - most IR utilities independent of IR level
- **Source level type information preserved throughout**
  - location operands type memory accesses
  - locals/temps typed (program point specific)
  - GC maps computed from local/temp type information
- **Guard operands encode control dependence as data dependence**
  - Link PEI's with guarded operations (null_check to load)
  - Work in progress: originally designed for local opts, being extended for global opts (issue: cond branches produce guards)

Mistakes

- Premature optimization (the root of all evil ...)
  - Early implementation overly biased towards compilation speed
  - Unnecessary avoidance of inheritance in IR
  - Didn't trust language features (interfaces, efficient GC, etc.)
- Should've designed compiler to be reentrant from the beginning
  - Difficult to reengineer after the fact
- Unsound data structures in initial IR implementation
  - We are still recovering
  - Better invariant checking needed
- Outstanding performance issues
  - Opt level 2 not so great
  - IA/32 floating point

#1 Issue in Building a Research Compiler

- Researchers are rewarded for writing "throw-away" code
  - Paper deadlines
  - No users to worry about
  - Don't have to worry about (un)reproducible results
- If the project is successful, "throw-away" code either
  - Lives forever, tormenting you until you can't take the pain, so you rewrite it much later at greater cost
  - Gets thrown away; doesn't benefit future users
- Throw-away code conflicts with good science and open source spirit

Jikes RVM Optimizing Compiler Summary of our Experience

- We built a portable and retargetable dynamic optimizing compiler for Java bytecodes from scratch
- We made some mistakes
- We learned some lessons
- Further enhancements from the community would be most welcome!
  - See list of suggested features on the Jikes RVM home page

www.ibm.com/developerworks/oss/jikesrvm

Jikes RVM Team

www.ibm.com/developerworks/oss/jikesrvm

Optimizing Compiler Developers

Matthew Arnold	David Grove	Janice Shepherd
Perry Cheng	Michael Hind	Harini Srinivasan
Jong-Deok Choi	Igor Pechtchanski	Peter F. Sweeney
Julian Dolby	Vivek Sarkar	Martin Trapp
Stephen Fink	Mauricio Serrano	John Whaley
	Arvin Shepherd	

Runtime Developers

Bowen Alpern
Dick Attanasio
David Bacon
Maria Butrico
Anthony Cocchi
Derek Lieber
Mark Mergen
Ton Ngo
Stephen Smith

Other Contributors

John Barton	Kim Hazelwood (co-op)	Eliot Moss (visitor)
Rastislav Bodik (visitor)	Martin Hirzel (co-op)	Anne Mulhern (co-op)
Rajesh Bordawekar	Susan Hummel	Jeff Palm (Colorado)
Steve Blackburn (visitor)	Chandra Krintz (co-op)	D.J. Penny (Intel)
Michael Burke	Han Lee (co-op)	Feng Qian (co-op)
John Cavazos (co-op)	John Leuner	Barbara G. Ryder (visitor)
Craig Chambers (visitor)	Keunwoo Lee (co-op)	Yefim Shuf (co-op)
Brian Cooper (co-op)	Vassily Litvinov (co-op)	Sharad Singhai (co-op)
Jeanne Ferrante (visitor)	Alexey Loginov (co-op)	V. C. Sreedhar
Tracy Ferguson (co-op)	Jan Maessen (co-op)	Manu Sridharan (co-op)
Chapman Flack (Purdue)	Kathryn McKinley (visitor)	Hyun-Gyoo Yook (co-op)
Manish Gupta	Sam Midkiff	Lingli Zhang (UCSB)

Jikes RVM (Jalapeño) References

Compiler Optimizations I

- [SAS'00] "Unified Analysis of Array and Object References in Strongly Typed Languages" by Fink, Knobe, and Sarkar, *2000 Static Analysis Symposium*, Santa Barbara, CA, June, 2000.
- [TOPLAS'99] "Linear Scan Register Allocation" by Poletto and Sarkar, *ACM Transactions on Programming Languages and Systems,* 21(5) Sept. 1999
- [PASTE'99] "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs" by Choi, Grove, Hind, and Sarkar, *1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, Sept. 1999.
- [LCPC'99] "Dependence Analysis for Java" by Chambers, Pechtchanski, Sarkar, Serrano, Srinivasan, *Workshop on Languages and Compilers for Parallel Computing*, La Jolla, CA, Aug 1999
- [PLDI'00] "ABCD: Eliminating Array Bounds Checks on Demand", by Bodik, Gupta, and Sarkar, *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

Jikes RVM (Jalapeño) References

Compiler Optimizations II

- [DYNAMO'00] "A Comparative Study of Static and Profile-Based Heuristics for Inlining" by Arnold, Fink, Sarkar, and Sweeney, *2000 ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Boston, Massachusetts, Jan 2000.
- [OOPSLA'99] "Escape Analysis for Java" by Choi, Gupta, Serrano, Sreedhar, Midkiff, *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov 1999.
- [PLDI'00] "A Framework for Interprocedural Analysis and Optimization in the Presence of Dynamic Class Loading" by Sreedhar, Burke, Choi, *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.
- [ECOOP'00] "Optimizing Java Programs in the Presence of Exceptions" by Gupta, Choi, Hind, 1*4th European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

Jikes RVM (Jalapeño) References

Adaptive Optimizations

- [OOSPLA'02] "Online Feedback-Directed Optimization of Java" by Arnold, Hind, and Ryder, *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, Minnesota, Oct 2000.
- [ECOOP'02] "Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading" by Arnold and Ryder, *European Conference on Object-Oriented Programming*, Malaga Spain, June 2002.
- [PLDI'01] "A Framework for Reducing the Cost of Instrumented Code" by Arnold and Ryder, *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
- [FDDO'00] "Adaptive Optimization in the Jalapeño JVM: The Controller's Analytical Model" by Arnold, Fink, Grove, Hind, and Sweeney, *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000.
- [OOPSLA'00] "Adaptive Optimization in the Jalapeño JVM" by Arnold, Fink, Grove, Hind, and Sweeney, *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, Minnesota, Oct 2000.
- [LCPC'00] "An Empirical Study of Selective Optimization" by Arnold, Hind, and Ryder, *13th International Workshop on Languages and Compilers for Parallel Computing*, Yorktown Heights, New York, Aug 2000.

Jikes RVM (Jalapeño) References

Overview

- [SJ'00] "The Jalapeño Virtual Machine", by Alpern et al., IBM System Journal, Vol 39, No 1, Feb 2000.
- [Grande'99] "The Jalapeño Dynamic Optimizing Compiler for Java" by Burke, Choi, Fink, Grove, Hind, Sarkar, Serrano, Sreedhar, Srinivasan, 1999 ACM Java Grande Conference, San Francisco, June 1999
- [WCSS'99] "Jalapeño - a Compiler-supported Java Virtual Machine for Servers" by Alpern, Cocchi, Lieber, Mergen, Sarkar, *Workshop on Compiler Support for Software System*, Atlanta, GA, May 1999.
- [JVM'02] "Experiences Porting the Jikes RVM to Linux/IA32" by Alpern, Butrico, Cocchi, Dolby, Fink, Grove, and Ngo, *2nd Java(TM) Virtual Machine Research and Technology Symposium*, San Francisco, California, Aug 2002.

Jikes RVM (Jalapeño) References

Runtime Services

- [ECOOP'02] "Space- and Time-Efficient Implementation of the Java Object Model" by Bacon, Fink, and Grove, *European Conference on Object-Oriented Programming*, Malaga Spain, June 2002.
- [OOPSLA'01] "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless", by Alpern, Cocchi, Fink, Grove, and Lieber, *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Tampa, FL, Oct 2001.
- [JVM'01] "Dynamic Type Checking in Jalapeño" by Alpern, Cocchi, and Grove, *The Usenix Java Virtual Machine Research and Technology Symposium*, Apr 2001.
- [OOPSLA'99] "Implementing Jalapeño in Java" by Alpern, Attanasio, Barton, Cocchi, Flynn Hummel, Lieber, Mergen, Ngo, Shepherd, and Smith, *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov 1999.

Jikes RVM (Jalapeño) References

Memory Management

- [LCPC'01] "A Comparative Evaluation of Parallel Garbage Collectors" by Attanasio, Bacon, Cocchi, and Smith, *Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, Kentucky, Aug, 2001.
- [PLDI'01] "Java without the Coffee Breaks: A Non-intrusive Multiprocessor Garbage Collector" by Bacon, Attanasio, Lee, Rajan, and Smith, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
- [ECOOP'01] "Concurrent Cycle Collection in Reference Counted Systems" by Bacon and Rajan, The Fifteenth European Conference on Object-Oriented Programming, University Eötvös Loránd, Budapest, Hungary, June 2001.

www.ibm.com/developerworks/oss/jikesrvm

Other Papers Cited

- [AWZ'88] "Detecting Equality of Variables in Programs" by Alpern, Wegman, Zadeck, POPL'88
- [Cohen'91] "Type-extension type tests can be performed in constant time" by Cohen, TOPLAS 13, No. 4.
- [Click'95] "Global Code Motion, Global Value Numbering" by Click, PLDI'95
- [Detlefs & Agesen '98] "Inlining of Virtual Methods" by Detlefs and Agesen, ECOOP'98
- [Fraser, Hanson, Proebsting '92] "Engineering, a simple, efficient code-generator generator" by Fraser, Hanson, and Proebsting, ACM LOPLAS 1(3), Sep. '92
- [Knobe & Sarkar '98] "Conditional Constant Propagation of Scalar and Array References Using Array SSA Form" by Knobe and Sarkar, SAS'98
- [Sarkar, Serrano, Simons '01] "Register-sensitive Selection, Duplication, and Sequencing of Instructions" by Sarkar, Serrano, Simons, ICS'01